

Programming Windows® 8 Apps with HTML, CSS, and JavaScript

**SECOND
PREVIEW**

Kraig Brockschmidt

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2012 Microsoft Corporation

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

ISBN: 978-0-7356-7261-1

This document supports a preliminary release of a software product that may be changed substantially prior to final commercial release. This document is provided for informational purposes only and Microsoft makes no warranties, either express or implied, in this document. Information in this document, including URL and other Internet website references, is subject to change without notice. The entire risk of the use or the results from the use of this document remains with the user.

Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions, Developmental, and Project Editor: Devon Musgrave

Cover: Twist Creative • Seattle

Introduction	11
Who This Book Is For	12
What You'll Need	13
A Formatting Note.....	13
Acknowledgements	13
Errata & Book Support	14
We Want to Hear from You.....	15
Stay in Touch	15
 Chapter 1: The Life Story of a WinRT App:	
Platform Characteristics of Windows 8.....	16
Leaving Home: Onboarding to the Store.....	17
Discovery, Acquisition, and Installation	20
Playing in Your Own Room: The App Container	23
Different Views of Life: View States and Resolution Scaling	27
Those Capabilities Again: Getting to Data and Devices	30
Taking a Break, Getting Some Rest: Process Lifecycle Management	33
Remembering Yourself: App State and Roaming	35
Coming Back Home: Updates and New Opportunities.....	39
And, Oh Yes, Then There's Design.....	40
 Chapter 2: Quickstart.....	42
A Really Quick Quickstart: The Blank App Template.....	42
Blank App Project Structure	45
QuickStart #1: Here My Am! and an Introduction to Blend for Visual Studio	50
Design Wireframes.....	50
Create the Markup	53
Styling in Blend	55
Adding the Code	59
Extra Credit: Receiving Messages from the iframe	71
The Other Templates	73
Fixed Layout Template.....	73
Navigation Template.....	74

Grid Template	74
Split Template	74
What We've Just Learned	75
Chapter 3: App Anatomy and Page Navigation	76
Local and Web Contexts within the App Host	77
Referencing Content from App Data: ms-appdata	81
Sequential Async Operations: Chaining Promises	84
Debug Output, Error Reports, and the Event Viewer	87
App Activation	89
Branding Your App 101: The Splash Screen and Other Visuals	89
Activation Event Sequence	92
Activation Code Paths	93
WinJS.Application Events	95
Extended Splash Screens	97
App Lifecycle Transition Events and Session State	99
Suspend, Resume, and Terminate	100
Basic Session State in Here My Am!	104
Data from Services and WinJS.xhr	106
Handling Network Connectivity (in Brief)	109
Tips and Tricks for WinJS.xhr	109
Page Controls and Navigation	111
WinJS Tools for Pages and Page Navigation	111
The Navigation App Template, PageControl Structure, and PageControlNavigator	112
The Navigation Process and Navigation Styles	118
Optimizing Page Switching: Show-and-Hide	120
Completing the Promises Story	120
What We've Just Learned	122
Chapter 4: Controls, Control Styling, and Data Binding	124
The Control Model for HTML, CSS, and JavaScript	125
HTML Controls	126

WinJS stylesheets: ui-light.css, ui-dark.css, and win-* styles.....	129
Extensions to HTML Elements	130
WinJS Controls	130
WinJS Control Instantiation.....	132
Strict Processing and processAll Functions	133
Example: WinJS.UI.Rating Control.....	134
Example: WinJS.UI.Tooltip Control.....	135
Working with Controls in Blend	137
Control Styling	139
Styling Gallery: HTML Controls	141
Styling Gallery: WinJS Controls	143
Some Tips and Tricks.....	146
Custom Controls.....	147
Custom Control Examples	149
Custom Controls in Blend.....	151
Data Binding	154
Data Binding in WinJS.....	157
Additional Binding Features.....	162
What We've Just Learned	165
Chapter 5: Collections and Collection Controls	167
Collection Control Basics	168
Quickstart #1: The HTML FlipView Control Sample	168
Quickstart #2a: The HTML ListView Essentials Sample.....	170
Quickstart #2b: The ListView Grouping Sample.....	172
ListView in the Grid App Project Template.....	177
The Semantic Zoom Control	181
FlipView Features and Styling	184
Data Sources	187
A FlipView Using the Pictures Library	187
Custom Data Sources	189

How Templates Really Work.....	191
Referring to Templates	191
Template Elements and Rendering.....	192
Template Functions (Part 1): The Basics.....	193
ListView Features and Styling.....	195
When Is ListView the Wrong Choice?	195
Options, Selections, and Item Methods	197
Styling.....	200
Backdrops.....	201
Layouts and Cell Spanning	202
Optimizing ListView Performance	208
Random Access	209
Incremental Loading.....	210
Template Functions (Part 2): Promises, Promises!	210
What We've Just Learned	216
Chapter 6: Layout.....	218
Principles of WinRT app Layout	219
Quickstart: Pannable Sections and Snap Points.....	223
Laying Out the Hub	223
Laying Out the Sections.....	225
Snap Points	225
The Many Faces of Your Display	226
View States.....	227
Screen Size, Pixel Density, and Scaling	234
Adaptive and Fixed Layouts for Display Size	238
Fixed Layouts and the ViewBox Control	239
Adaptive Layouts	241
Using the CSS Grid.....	243
Overflowing a Grid Cell	244
Centering Content Vertically	245

Scaling Font Size	246
Item Layout	247
CSS 2D and 3D Transforms.....	247
Flexbox	248
Nested and Inline Grids	249
Fonts and Text Overflow	250
Multicolumn Elements and Regions.....	251
What We've Just Learned	254
Chapter 7: Commanding UI	256
Where to Place Commands	257
The App Bar	261
App Bar Basics and Standard Commands	263
App Bar Styling	272
Command Menus	274
Custom App Bars and Navigation Bars	276
Flyouts and Menus.....	277
WinJS.UI.Flyout Properties, Methods, and Events.....	279
Flyout Examples	280
Menus and Menu Commands.....	283
Message Dialogs.....	288
Improving Error Handling in Here My Am!	289
What We've Just Learned	294
Chapter 8: State, Settings, Files, and Documents	295
The Story of State.....	296
Settings and State.....	298
App Data Locations.....	299
AppData APIs (WinRT and WinJS).....	301
Using App Data APIs for State Management	310
Settings Pane and UI	316
Design Guidelines for Settings	317

Populating Commands	320
Implementing Commands: Links and Settings Flyouts	321
User Data: Libraries, File Pickers, and File Queries	326
Using the File Picker	327
Media Libraries	336
Documents and Removable Storage.....	337
Rich Enumeration with File Queries.....	338
Here My Am! Update.....	344
What We've Just Learned	344
Chapter 9: Input and Sensors	346
Touch, Mouse, and Stylus Input	347
The Touch Language, Its Translations, and Mouse/Keyboard Equivalents	348
What Input Capabilities Are Present?	355
Unified Pointer Events.....	357
Gesture Events	360
The Gesture Recognizer	369
Keyboard Input and the Soft Keyboard.....	371
Soft Keyboard Appearance and Configuration	371
Adjusting Layout for the Soft Keyboard	374
Standard Keystrokes	376
Inking	377
Geolocation.....	380
Sensors.....	383
What We've Just Learned	386
Chapter 10: Media	387
Creating Media Elements	388
Graphics Elements: Img, Svg, and Canvas (and a Little CSS)	390
Additional Characteristics of Graphics Elements.....	393
Some Tips and Tricks.....	394
Video Playback and Deferred Loading	398

Disabling Screen Savers and the Lock Screen During Playback.....	400
Video Element Extension APIs.....	401
Applying a Video Effect.....	402
Browsing Media Servers	403
Audio Playback and Mixing.....	403
Audio Element Extension APIs.....	405
Playback Manager and Background Audio	406
Playing Sequential Audio.....	410
Playlists	411
Loading and Manipulating Media.....	414
Media File Metadata.....	414
Image Manipulation and Encoding.....	423
Manipulating Audio and Video	429
Media Capture	433
Flexible Capture with the MediaCapture Object.....	435
Selecting a Media Capture Device.....	439
Streaming Media and PlayTo.....	440
Streaming from a Server and Digital Rights Management (DRM)	441
Streaming from App to Network	442
PlayTo.....	443
What We Have Learned	446
Chapter 11: Purposeful Animations	448
Systemwide Enabling and Disabling of Animations.....	450
The WinJS Animations Library	451
Animations in Action.....	454
CSS Animations and Transitions.....	458
The Independent Animations Sample	463
Rolling Your Own: Tips and Tricks.....	464
What We've Just Learned	469
Chapter 12: Contracts.....	470

Share.....	472
Source Apps.....	474
Target Apps	480
The Clipboard	491
Search.....	493
Search in the App Manifest and the Search Item Template.....	496
Basic Search and Search Activation	496
Providing Query Suggestions	499
Providing Result Suggestions	503
Type to Search.....	504
Launching Apps: File Type and URI Scheme Associations.....	504
File Activation	506
Protocol Activation	508
File Picker Providers	509
Manifest Declarations.....	510
Activation of a File Picker Provider.....	511
Cached File Updater.....	518
Updating a Local File: UI	521
Updating a Remote File: UI	522
Update Events.....	523
Contacts	527
Using the Contact Picker.....	529
Contact Picker Providers	531
What We've Just Learned	534
About the Author	536
Survey.....	537

Introduction

Welcome, my friends, to Windows 8! On behalf of the thousands of designers, program managers, developers, test engineers, and writers who have brought the product to life, I'm delighted to welcome you into a world of **Windows Reimagined**.

This theme is no mere sentimental marketing ploy, intended to bestow an aura of newness to something that is essentially unchanged, like those household products that make a big splash on the idea of "New and Improved *Packaging!*" No, Microsoft Windows truly has been reborn—after more than a quarter-century, something genuinely new has emerged.

I suspect—indeed expect—that you're already somewhat familiar with the reimagined user experience of Windows 8. You're probably reading this book, in fact, because you know that the ability of Windows 8 to reach across desktop, laptop, and tablet devices, along with the global reach of the Windows Store, will provide you with tremendous business opportunities, whether you're in business, as I like to say, for fame, fortune, fun, or philanthropy.

We'll certainly see many facets of this new user experience throughout the course of this book. Our primary focus, however, will be on the reimagined *developer* experience.

I don't say this lightly. When I first began giving presentations within Microsoft about building WinRT apps, as they are called (and also referred to as Windows Store apps in consumer contexts), I liked to show a slide of what the world was like in the year 1985. It was the time of Ronald Reagan, Margaret Thatcher, and Cold War tensions. It was the time of VCRs and the discovery of AIDS. It was when *Back to the Future* was first released, Michael Jackson topped the charts with *Thriller*, and Steve Jobs was kicked out of Apple. And it was when software developers got their first taste of the original Windows API and the programming model for desktop applications.

The longevity of that programming model has been impressive. It's been in place for over a quarter-century now and has grown to become the heart of the largest business ecosystem on the planet. The API itself, known today as Win32, has also grown to become the largest on the planet! What started out on the order of about 300 callable methods has expanded three orders of magnitude, well beyond the point that any one individual could even hope to understand a fraction of it. I'd certainly given up such futile efforts myself.

So when I bumped into my old friend Kyle Marsh in the fall of 2009 just after Windows 7 had been released and heard from him that Microsoft was planning to reinvigorate native app development for Windows 8, my ears were keen to listen. In the months that followed I learned that Microsoft was introducing a completely new API called the Windows Runtime (or WinRT). This wasn't meant to replace Win32, mind you; desktop applications would still be supported. No, this was a programming model built from the ground up for a new breed of touch-centric, immersive apps that could compete with those emerging on various mobile platforms. It would be designed from the app developer's

point of view, rather than the system's, so that key features would take only a few lines of code to implement rather than hundreds or thousands. It would also enable direct native app development in multiple programming languages, which meant that new operating system capabilities would surface to those developers without having to wait for an update to some intermediate framework.

This was very exciting news to me because the last time that Microsoft did anything significant to the Windows programming model was in the early 1990s with a technology called the Component Object Model (COM), which is exactly what allowed the Win32 API to explode as it did. Ironically, it was my role at that time to introduce COM to the developer community, which I did through two editions of *Inside OLE* (Microsoft Press) and seemingly endless travel to speak at conferences and visit partner companies. History, indeed, does tend to repeat itself, for here I am again!

In December 2010, I was part of small team who set out to write the very first WinRT apps using what parts of the new WinRT API had become available. Notepad was the text editor of choice, we built and ran apps on the command line by using abstruse Powershell scripts that required us to manually type out ungodly hash strings, we had no documentation other than functional specifications, and we basically had no debugger to speak of other than the tried and true `document.write()`. Indeed, we generally worked out as much HTML, CSS, and JavaScript as we could inside a browser with F12 debugging tools, only adding WinRT-specific code at the end because browsers couldn't resolve those APIs. You can imagine how we celebrated when we got anything to work at all!

Fortunately, it wasn't long before tools like Visual Studio Express and Blend for Visual Studio became available. By the spring of 2011, when I was giving many training sessions to people inside Microsoft on building WinRT apps, the process was becoming far more enjoyable and far, far more productive. Indeed, while it took us some weeks in late 2010 to get even Hello World to show up on the screen, by the fall of 2011 we were working with partner companies who pulled together complete Store-ready apps in roughly the same amount of time.

As we've seen—thankfully fulfilling our expectations—it's possible to build a great WinRT app in a matter of weeks. I'm hoping that this present volume, along with the extensive resources on <http://dev.windows.com>, will help you to accomplish exactly that and to reimagine your own designs.

Who This Book Is For

This book is about writing WinRT apps using HTML5, CSS3, and JavaScript. Our primary focus will be on applying these web technologies within the Windows 8 platform, where there are unique considerations, and not on exploring the details of those web technologies themselves. For the most part, then, I'm assuming that you're already at least somewhat conversant with these standards. We will cover some of the more salient areas like the CSS grid, which is central to app layout, but otherwise I trust that you're capable of finding appropriate references for everything else.

I'm also assuming that your interest in Windows 8 has at least two basic motivations. One, you

probably want to come up to speed as quickly as you can, perhaps to carve out a foothold in the Windows Store sooner rather than later. Toward that end, I've front-loaded the early chapters with the most important aspects of app development along with "Quickstart" sections to give you immediate experience with the tools, the API, and core platform features. On the other hand, you probably also want to make the best app you can, one that performs really well and that takes advantage of the full extent of the platform. Toward this end, I've also endeavored to make this book comprehensive, helping you at least be aware of what's possible and where optimizations can be made.

Many insights have come from working directly with real-world developers on their real-world apps. As part of the Windows Ecosystem team, myself and my teammates have been on the front lines bringing those first apps to the Windows Store. This has involved writing bits of code for those apps and investigating bugs, along with conducting design, code, and performance reviews with members of the core Windows engineering teams. As such, one of my goals with this book is to make that deep understanding available to many more developers, including you!

What You'll Need

To work through this book, you should download and install the latest developer build of Windows 8 along with the Windows SDK and tools. These, along with a number of other resources, are listed on <http://msdn.microsoft.com/en-us/windows/apps/br229516>. I also recommend you visit <http://code.msdn.microsoft.com/windowsapps/Windows-8-Modern-Style-App-Samples> and download the entire set of JavaScript samples; we'll be using many of them throughout this book.

A Formatting Note

Throughout this book, identifiers that appear in code, such as variable names, property names, and API functions and namespaces, are formatted with a color and a fixed-point font. Here's an example: `Windows.Storage.ApplicationData.current`. At times these fully qualified names—those that include the entire namespace—can become quite long, they are sometimes hyphenated across line breaks, as in `Windows.Security.Cryptography.CryptographicBuffer.convertStringToBinary`. Generally speaking, I've tried to hyphenate after a dot or between combined words but not within a word (and, as you can see earlier in this paragraph, this doesn't always work out). In any case, these hyphens are never part of the identifier except when used in CSS where hyphens are allowed.

Acknowledgements

In many ways, this isn't *my* book—that is, it's not an account of my own experiences and opinions about WinRT apps on Windows 8. I'm serving more as a storyteller, where the story itself has been written by the thousands of people in the Windows team whose passion and dedication have been a

constant source of inspiration. Writing a book like this wouldn't be possible without all the work that's gone into customer research, writing specs, implementing, testing, and documenting all the details, managing daily builds and public releases, and writing perhaps the best set of samples I've ever seen for a platform. We'll be drawing on many of those samples, in fact, and even the words in some sections come directly from conversations I've had with the people who designed and developed a particular feature. I'm grateful for their time, and I'm delighted to give them a voice through which they can share their passion for excellence with you.

A number of individuals deserve special mention for their long-standing support of this project. First to Chris Sells, with whom I co-authored the earliest versions of this book; to Mahesh Prakriya, Ian LeGrow, Anantha Kancherla, Keith Boyd and their respective teams, with whom I've worked closely; and to Keith Rowe, Dennis Flanagan, and Ulf Schoo, under whom I've had the pleasure of serving. Thanks also to Devon Musgrave at Microsoft Press, and to all those who have reviewed chapters and provided answers to my endless streams of questions: Chris Tavares, Jesse McGatha, Josh Williams, Feras Moussa, Jake Sabulsky, Henry Tappen, David Tepper, Mathias Jourdain, Ben Betz, Ben Srouer, Adam Barrus, Ryan Demopoulos, Sam Spencer, Damian Kedzierski, Bill Ticehurst, Tarek Anya, Scott Graham, Scott Dickens, Jerome Holman, Kenichiro Tanaka, Sean Hume, Patrick Dengler, David Washington, Scott Hoogerwerf, Harry Pierson, Jason Olson, Justin Cooperman, Rohit Pagariya, Nathan Kuchta, Kevin Woley, Markus Mielke, Paul Gusmorino, Marc Wautier, Charing Wong, Chantal Leonard, Vincent Celie, Edgar Ruiz Silva, Mike Mastrangelo, Derek Gephard, Tyler Beam, Adam Stritzel, Rian Chung, Shijun Sun, Dale Rogerson, Megan Bates, Raymond Chen, Perumaal Shanmugam, Michael Crider, Axel Andrejs, Jake Sabulsky, as well as those I've forgotten and those still to come as the last set of chapters are added to the final edition. My direct teammates, Kyle Marsh, Todd Landstad, Shai Hinitz, Lora Heiny, and Joseph Ngari have also been invaluable in sharing what they've learned in working with real-world partners.

Finally, special hugs to my wife Kristi and our young son Liam, who have lovingly been there the whole time and who don't mind my traipsing through the house to my office either late at night or early in the morning.

Errata & Book Support

We've made every effort to ensure the accuracy of this preview ebook and its companion content. When the final version of this book is available (in fall 2012), any errors that are reported after the book's publication will be listed on our Microsoft Press site at oreilly.com. At that point, you can search for the book at <http://microsoftpress.oreilly.com> and then click the "View/Submit Errata" link. If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software is not offered through the addresses above. Support for developers, however, can be found on the Windows Developer Center's [support section](#).

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at

<http://www.microsoft.com/learning/booksurvey>

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: <http://twitter.com/MicrosoftPress>

Chapter 1

The Life Story of a WinRT App: Platform Characteristics of Windows 8

Paper or plastic? Fish or cut bait? To be or not to be? Standards-based or native? These are the questions of our time....

Well, OK, maybe most of these aren't the grist for university-level philosophy courses, but certainly the last one has been increasingly important for app developers. Standards-based apps are great because they run on multiple platforms; your knowledge and experience with standards like HTML5 and CSS3 are likewise portable. Unfortunately, because standards generally take a long time to produce, they always lag behind the capabilities of the platforms themselves. After all, competing platform vendors will, by definition, always be trying to differentiate! For example, while HTML5 now has a standard for geolocation/GPS sensors and has started on working drafts for other forms of sensor input (like accelerometers, compasses, near-field proximity, and so on), native platforms already make these available. And by the time HTML's standards are in place and widely supported, the native platforms will certainly have added another set of new capabilities.

As a result, developers wanting to build apps around cutting-edge features—to differentiate from their own competitors!—must adopt the programming language and presentation technology imposed by each native platform or take a dependency on a third-party framework that tries to bridge the differences.

Bottom line: it's a hard choice.

Fortunately, Windows 8 provides what I personally think is a brilliant solution for apps. Early on, the Windows team set out to solve the problem of making native capabilities—the system API, in other words—*directly* available to *any* number of programming languages, including JavaScript. This is what's known as the Windows Runtime API, or just *WinRT* for short.

WinRT APIs are implemented according to a certain low-level structure and then “projected” into different languages in a way that looks and feels natural to developers familiar with those languages. This includes how objects are created, configured, and managed; how events, errors, and exceptions are handled; how asynchronous operations work (to keep the user experience fast and fluid); and even the casing of names on methods, properties, and events.

The Windows team also made it possible to write native apps that employ a variety of presentation technologies, including DirectX, XAML, and, in the case of apps written in JavaScript, HTML5 and CSS3.

This means that Windows gives you—a developer already versed in HTML, CSS, and JavaScript standards—the ability to *use what you know* to write fully native Windows 8 apps using the WinRT API. Those apps will, of course, be specific to the Windows 8 platform, but the fact that you don't have to learn a completely new programming paradigm is worthy of taking a week off to celebrate—especially because you won't have to spend that week (or more) learning a complete new programming paradigm!

Throughout this book we'll explore how to leverage what you know of standards-based web technologies to build great Windows 8 apps. In the next chapter we'll focus on the basics of a working app and the tools used to build it. Then we'll look at fundamentals like the fuller anatomy of an app, controls, collections, layout, commanding, state management, and input, followed by chapters on media, animations, contracts through which apps work together, networking, devices, WinRT components, and the Windows Store (a topic that includes localization and accessibility). There is much to learn.

For starters, let's talk about the environment in which apps run and the characteristics of the platform on which they are built—especially the terminology that we'll depend on in the rest of the book (highlighted in *italics*). We'll do this by following an app's journey from the point when it first leaves your hands, through its various experiences with your customers, to where it comes back home for rest, renewal, and rebirth. For in many ways your app is like a child: you nurture it through all its formative stages, doing everything you can to prepare it for life in the great wide world. So it helps to understand the nature of that world!

Terminology note What we refer to as *WinRT apps* are those that are acquired from the Windows Store and for which all the platform characteristics in this chapter (and book) apply. In consumer contexts these are also known as Windows Store apps, but since we're primarily interested in how they're written—using the WinRT API—we'll refer to them as WinRT apps. These are distinctly different from traditional *desktop applications* that are acquired through regular retail channels and installed through their own installer programs. Unless noted, then, an "app" in this book refers to a WinRT app.

Leaving Home: Onboarding to the Store

For WinRT apps, there's really one port of entry into the world: customers always acquire, install, and update apps through the *Windows Store*. Developers and enterprise users can side-load apps, but for the vast majority of the people you care about, they go to the Windows Store and the Store alone.

This obviously means that an app—the culmination of your development work—has to get into the Store in the first place. This happens when you take your pride and joy, package it up, and upload it to the Store by using the Store/Upload App Package command in Visual Studio.¹ The *package* itself is an

¹ To do this you'll need to create a developer account with the Store by using the Store/Open Developer Account command in Visual Studio Express. Visual Studio Express and Expression Blend, which we'll be using as well, are free tools that you can obtain

appx file (.appx)—see Figure 1-1—that contains your app’s code, resources, libraries, and a *manifest*. The manifest describes the app (names, logos, etc.), the *capabilities* it wants to access (such as areas of the file system or specific devices like cameras), and everything else that’s needed to make the app work (such as file associations, declaration of background tasks, and so on). Trust me, we’ll become great friends with the manifest!

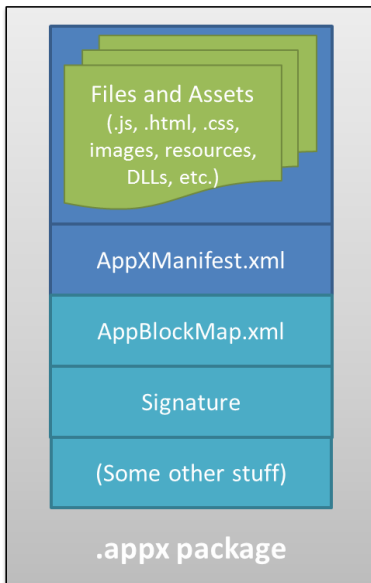


FIGURE 1-1 An appx package is simply a zip file that contains the app’s files and assets, the app manifest, a signature, and a sort of table-of-contents called the blockmap. When uploading an app, the initial signature is provided by Visual Studio; the Windows Store will re-sign the app once it’s certified. The blockmap, for its part, describes how the app’s files are broken up into 64K blocks. In addition to providing certain security functions (like detecting whether a package has been tampered with) and performance optimization, the blockmap is used to determine exactly what parts of an app have been updated between versions so the Windows Store only needs to download those specific blocks rather than the whole app anew.

The upload process will walk you through setting your app’s name, choosing selling details (including price tier, in-app purchases, and trial periods), providing a description and graphics, and also providing notes to manual testers. After that, your app essentially goes through a series of job interviews, if you will: background checks (malware scans and GeoTrust certification) and manual testing by a human being who will read the notes you provide (so be courteous and kind!). Along the way you can check your app’s progress through the [Windows Store Dashboard](#).²

from <http://dev.windows.com>. This also works in Visual Studio Ultimate, the fuller, paid version of this flagship development environment.

² All of the automated tests except the malware scans are incorporated into the Windows App Certification Kit, affectionately known as the WACK. This is part of the Windows SDK that is itself included with the Visual Studio Express/Expression Blend

The overarching goal with these job interviews (or maybe it's more like getting through airport security!) is to help users feel confident and secure in trying new apps, a level of confidence that isn't generally found with apps acquired from the open web. As all apps in the Store are certified, signed, and subject to ratings and reviews, customers can trust all apps from the Store as they would trust those recommended by a reliable friend. Truly, this is wonderful news for most developers, especially those just getting started—it gives you the same access to the worldwide Windows market that has been previously enjoyed only by those companies with an established brand or reputation.

It's worth noting that because you set up pricing, trial versions, and in-app purchases during the on-boarding process, you'll have already thought about your app's relationship to the Store quite a bit! After all, the Store is where you'll be doing business with your app, whether you're in business for fame, fortune, fun, or philanthropy.

As a developer, indeed, this relationship spans the entire lifecycle of an app—from planning and development to distribution, support, and servicing. This is, in fact, why I've started this life story of an app with the Windows Store, because you really want to understand that whole lifecycle from the very beginning of planning and design. If, for example, you're looking to turn a profit from a paid app or in-app purchases, perhaps also offering a time-limited or feature-limited trial, you'll want to engineer your app accordingly. If you want to have a free, ad-supported app, or if you want to use a third-party commerce solution for in-app purchases (bypassing revenue sharing with the Store), these choices also affect your design from the get-go. And even if you're just going to give the app away to promote a cause or to just share your joy, understanding the relationship between the Store and your app is still important. (For all these reasons, you might want to skip ahead read the first parts of Chapter 17, "Apps for Everyone," before you start writing your app in earnest.)

Anyway, if your app hits any bumps along the road to certification, you'll get a report back with all the details, such as any violations of the [Certification requirements for Windows apps](#) (part of the [Windows Store agreements](#) section). Otherwise, congratulations—your app is ready for customers!

Sidebar: The Store API and Product Simulator

The `Windows.ApplicationModel.Store.CurrentProduct` class in WinRT provides the ability for apps to retrieve their product information from the store (including in-app purchases), check license status, and prompt the user to make purchases (such as upgrading a trial or making an in-app purchase).

Of course, this begs a question: how can an app test such features before it's even in the Store? The answer is that during development, you use these APIs through the `Windows.ApplicationModel.Store.CurrentProductSimulator` class instead. This is entirely identical to `CurrentProduct` except that it works against local data in an XML file rather than live Store data in the cloud. This allows you to simulate the various conditions that your app might

download. If you can successfully run the WACK during your development process, you shouldn't have any problem passing the first stage of onboarding.

encounter so that you can exercise all your code paths appropriately. Just before packaging your app and sending it to the Store, you just change `CurrentProductSimulator` to `CurrentProduct` and you're good to go. (If you forget, the simulator will simply fail on a non-developer machine, like those used by the Store testers.)

Discovery, Acquisition, and Installation

Now that your app is out in the world, its next job is to make itself known and attractive to potential customers. Simply said, while consumers can find your app in the Windows Store through browsing or search, you'll still need to market your product as always. That's one reality of the platform that certainly hasn't changed. That aside, even when your app is found in the Store it still needs to present itself well to its suitors.

Each app in the Store has a *product description page* where people see your app description, screen shots, ratings and reviews, and the capabilities your app has declared in its manifest, as shown in Figure 1-2. That last bit means you want to be judicious in declaring your capabilities. A music player app, for instance, will obviously declare its intent to access the user's music library but usually doesn't need to declare access to the pictures library unless it explains itself. Similarly, a communications app would generally ask for access to the camera and microphone, but a news reader app probably wouldn't. On the other hand, an ebook reader might declare access to the microphone *if* it had a feature to attach audio notes to specific bookmarks.

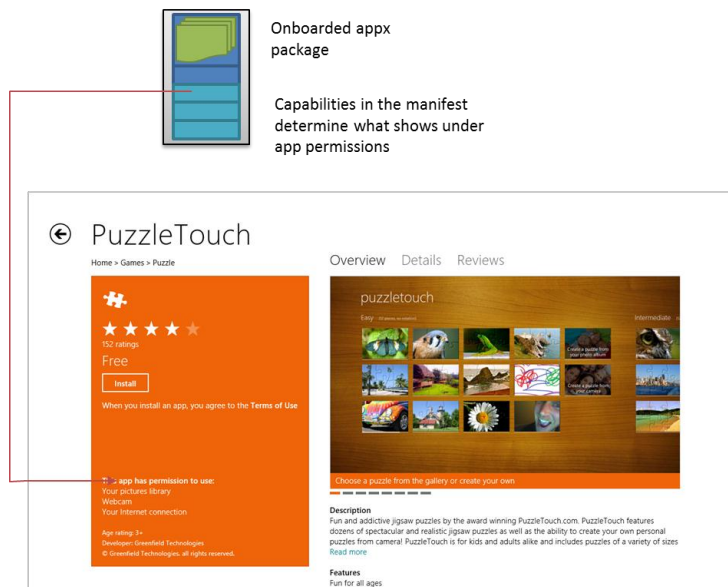
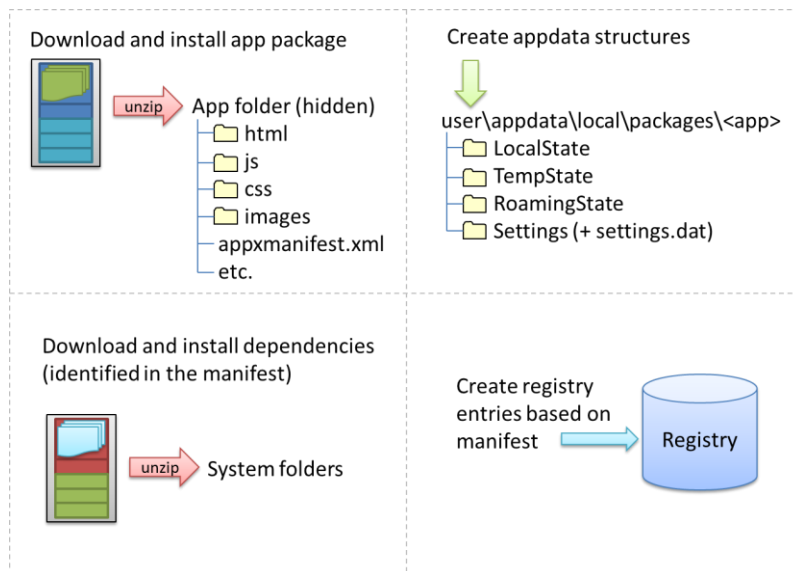


FIGURE 1-2 A typical app page in the Windows Store, where the manifest in the app package determines what appears in the app permissions. Here, for example, PuzzleTouch’s manifest declares the Pictures Library, Webcam, and Internet (Client) capabilities.

The point here is that what you declare needs to make sense to the user, and if there are any doubts you should clearly indicate the features related to those declarations in your app’s description. (Note how Puzzle Touch does that for the camera.) Otherwise the user might really wonder just what your news reader app is going to do with the microphone and might opt for another app that seems less intrusive.³

The user will also see your app pricing, of course, and whether you offer a trial period. Whatever the case, if they choose to install the app (getting it for free, paying for it, or accepting a trial), your app now becomes fully incarnate on a real user’s device. The appx package is downloaded to the device and installed automatically along with any dependencies, such as the *Windows Library for JavaScript* (see the sidebar on the next page.) As shown in Figure 1-3, the Windows deployment manager creates a folder for the app, extracts the package contents to that location, creates *appdata* folders (local, roaming, and temp, which the app can freely access, along with settings files for key-value pairs and some other system-managed folders), and does any necessary fiddling with the registry to install the app’s tile on the *Start screen*, create file associations, install libraries, and do all those other things that are again described in the manifest. There are no user prompts during this process—especially not those annoying dialogs about reading the licensing agreement!



³ The user always has the ability to disallow access to sensitive resources at run time for those apps that have declared the intent, as we’ll see later. However, as those capabilities surface directly in the Windows Store, you want to be careful to not declare those that you don’t really need.

FIGURE 1-3 The installation process for WinRT apps acquired from the Windows Store; the exact sequence is unimportant.

In fact, licensing terms are integrated into the Store; acquisition of an app implies acceptance of those terms. (However, it is perfectly allowable for apps to show their own license acceptance page on startup, as well as require an initial login to a service if applicable.) But here's an interesting point: do you remember the real purpose of all those lengthy, annoyingly all-caps licensing agreements that we all pretend to read? Almost all of them basically say that you can install the software on only one machine. Well, that changes with WinRT apps: instead of being licensed to a machine, they are licensed *to the user*, giving that user the right to install the app on up to five different devices.

In this way WinRT apps are a much more *personal* thing than desktop apps have traditionally been. They are less general-purpose tools that multiple users share and more like music tracks or other media that really personalize the overall Windows experience. So it makes sense that users can replicate their customized experiences across multiple devices, something that Windows supports through automatic roaming of app data and settings between those devices. (More on that later.)

In any case, the end result of all this is that the app and its necessary structures are wholly ready to awaken on a device, as soon as the user taps a tile on the Start page or launches it through features like Search and Share. And because the system knows about everything that happened during installation, it can also completely reverse the process for a 100% clean uninstall—completely blowing away the appdata folders, for example, and cleaning up anything and everything that was put in the registry. This keeps the rest of the system entirely clean over time, even though the user may be installing and uninstalling hundreds or thousands of apps. We like to describe this like the difference between having guests in your house and guests in a hotel. In your house, guests might eat your food, rearrange the furniture, break a vase or two, feed the pets leftovers, stash odds and ends in the backs of drawers, and otherwise leave any number of irreversible changes in their wake (and you know desktop apps that do this, I'm sure!). In a hotel, on the other hand, guests have access only to a very small part of the whole structure, and even if they trash their room, the hotel can clean it out and reset everything as if the guest was never there.

Sidebar: What Is the Windows Library for JavaScript?

The HTML, CSS, and JavaScript code in a WinRT app is only parsed, compiled, and rendered at run time. (See the “Playing in Your Own Room: The App Container” section below.) As a result, a number of system-level features for WinRT apps written in JavaScript, like controls, resource management, and default styling, are supplied through the Windows Library for JavaScript, or *WinJS*, rather than through the Windows Runtime API. This way, JavaScript developers see a natural integration of those features into the environment they already understand, rather than being forced to use different kinds of constructs.

WinJS, for example, provides an HTML implementation of a number of controls such that they appear as part of the DOM and can be styled like any other intrinsic HTML controls. This is much more natural for developers to work with than having to create an instance of some WinRT class,

bind it to an HTML element, and style it through code or some other markup scheme rather than CSS. Similarly, WinJS provides an animations library built on CSS, rather than forcing developers to learn some other structure to accomplish the same end. In both cases, WinJS provides a core implementation of the Windows 8 user experience so that apps don't have to figure out how to re-create that experience themselves.

Generally speaking, WinJS is a *toolkit* that contains a number of independent capabilities that can be used together or separately. So WinJS also provides helpers for common JavaScript coding patterns, simplifying the definition of namespaces and object classes, handling of asynchronous operations (that are all over WinRT) through *promises*, and providing structural models for apps, data binding, and page navigation. At the same time, it doesn't attempt to wrap WinRT unless there is a compelling scenario where WinJS can provide real value. After all, the mechanism through which WinRT is projected into JavaScript already translates WinRT structures into those familiar to JavaScript developers.

All in all, WinJS is essential for and shared between every WinRT app written in JavaScript, and it's automatically downloaded and updated as needed when dependent apps are installed. We'll see many of its features throughout this book.

Sidebar: Third-Party Libraries

WinJS is an example of a special shared library package that is automatically downloaded from the Windows Store for dependent apps. Microsoft maintains a few of these in the Store so that the package need be downloaded only once and then shared between apps. Shared third-party libraries are not currently supported.

However, apps can freely use third-party libraries by bringing them into their own app package, provided of course that the libraries use only the APIs available to WinRT apps. For example, apps written in JavaScript can certainly use jQuery, Modernizer, Dojo, prototype.js, Box2D, and others, with the caveat that some functionality, especially UI and script injection, might not be supported. Apps can also use third-party binaries—known as WinRT components—that are again included in the app package. Also see the "Hybrid Apps" sidebar later in this chapter.

Playing in Your Own Room: The App Container

Now just as the needs of each day may be different when we wake up from our night's rest, WinRT apps can wake up—be activated—for any number of reasons. The user can, of course, tap or click the app's tile on the Start page. An app can also be launched in response to charms like Search and Share, through file or protocol associations, and a number of other mechanisms. We'll explore these variants as we progress through this book. But whatever the case, there's a little more to this part of the story

for apps written in JavaScript.

In the app's hidden package folder are the same kind of source files that you see on the web: .html files, .css files, .js files, and so forth. These are not directly executable like .exe files for apps written in C#, Visual Basic, or C++, so something has to take those source files and produce a running app with them. When your app is activated, then, what actually gets launched is that something: a special *app host* process called `wwahost.exe`⁴, as shown in Figure 1-4.

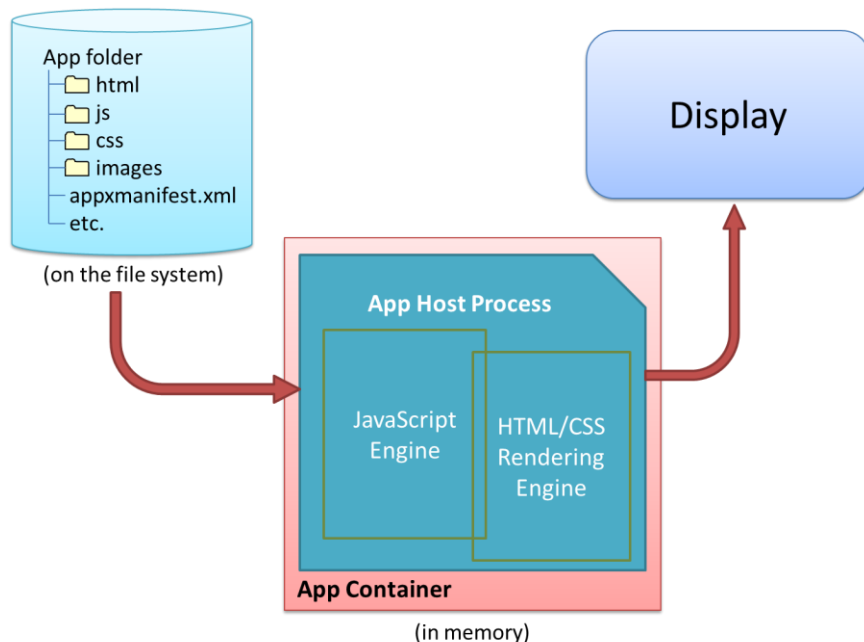


FIGURE 1-4 The app host is an executable (`wwahost.exe`) that loads, renders, and executes HTML, CSS, and JavaScript, in much the same way that a browser runs a web application.

The app host is more or less Internet Explorer 10 without the browser chrome—more in that your app runs on top of the same HTML/CSS/JavaScript engines as Internet Explorer, less in that a number of things behave differently in the two environments. For example:

- A number of methods in the DOM API are either modified or not available, depending on their design and system impact. For example, functions that display modal UI and block the UI thread are not available, like `window.alert`, `window.open`, and `window.prompt`. (Try `Windows.UI.Popups.MessageDialog` instead for some of these needs.)
- The engines support additional methods, properties, and even CSS media queries that are specific to being an app as opposed to a website. For example, special media queries apply to

⁴ “wwa” is an old acronym for WinRT apps written in JavaScript; some things just stick....

the different Windows 8 *view states*; see the next section. Elements like [audio](#), [video](#), and [canvas](#) also have additional methods and properties. (At the same time, objects like [MSApp](#) and methods like [requestAnimationFrame](#) that are available in Internet Explorer are also available to WinRT apps.)

- The default page of a WinRT app written in JavaScript runs in what's called the *local context* wherein JavaScript code has access to WinRT, can make cross-domain XMLHttpRequests, and can access remote media (videos, images, etc.). However, you cannot load remote script (from [http\[s\]://](#) sources, for example),⁵ and script is automatically filtered out of anything that might affect the DOM and open the app to injection attacks (e.g., [document.write](#) and [innerHTML](#) properties).
- Other pages in the app, as well as individual [iframe](#) elements within a local context page, can run in the *web context* wherein you get web-like behavior (such as remote script) but don't get WinRT access nor cross-domain XHR (though you can use parts of WinJS that don't rely on WinRT). Web context [iframes](#) are generally used to host web controls on a locally packaged page (like a map), as we'll see in Chapter 2, "Quickstart," or to load pages that are directly hosted on the web, while not allowing web pages to drive the app.

For full details, see [HTML and DOM API changes list](#) and [HTML, CSS, and JavaScript features and differences](#) on the Windows Developer Center, <http://dev.windows.com>. As with the app manifest, you should become good friends with the Developer Center.

Now all WinRT apps, whether hosted or not, run inside an environment called the *app container*. This is an insulation layer, if you will, that blocks local interprocess communication and either blocks or *brokers* access to system resources. The key characteristics of the app container are described next and then illustrated in Figure 1-5:

- All WinRT apps (other than those that are built into Windows) run within a dedicated environment that cannot interfere with or be interfered with other apps, nor can apps interfere with the system.
- WinRT apps, by default, get unrestricted read/write access only to their specific appdata folders on the hard drive (local, roaming, and temp). Access to everything else in the file system (including removable storage) has to go through a broker. This gatekeeper, if you will, provides access only if the app has declared the necessary capabilities in its manifest and/or the user has specifically allowed it. (We'll see the specific list of capabilities shortly.)
- WinRT apps cannot directly launch other apps by name or file path; they can programmatically launch other apps through file or URI scheme associations. As these are ultimately under the user's control, there's no guarantee that such an operation will start a specific app. However, we do encourage app developers to use app-specific URI schemes that will effectively identify your

⁵ Note that it is allowable in the local context to eval JavaScript code obtained from remote sources through other means, such as XHR. The restriction on directly loaded remote script is to specifically prevent cross-site scripting attacks.

specific app as a target. Technically speaking, another app could come along and register the same URI scheme (thereby giving the user a choice), but this is unlikely with a URI scheme that's closely related to the app's identity.

- Access to sensitive devices (like the camera, microphone, and GPS) is similarly controlled—the WinRT APIs that work with those devices will simply fail if the broker blocks those calls. And access to critical system resources, such as the registry, simply isn't allowed at all.
- WinRT apps are isolated from one another to protect from various forms of attack. This also means that some legitimate uses (like a snipping tool to copy a region of the screen to the clipboard) cannot be written as a WinRT app (so they must be a desktop application).
- Direct interprocess communication between WinRT apps, between WinRT apps and desktop applications, and between WinRT apps and local services, is blocked. Apps can still communicate through the cloud (web services, sockets, etc.), and many common tasks that require cooperation between apps—such as Search and Share—are handled through *contracts* in which those apps don't need to know any details about each other.

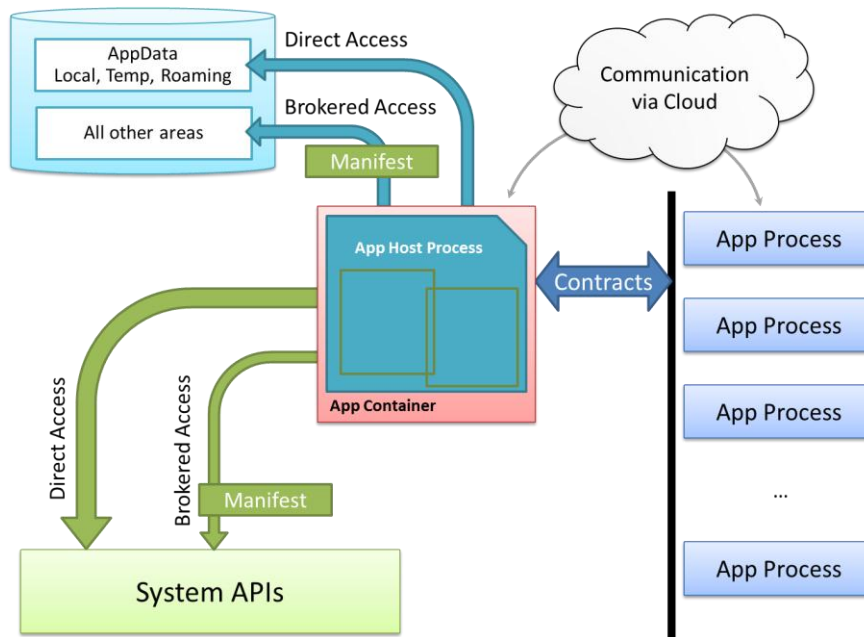


FIGURE 1-5 Process isolation for WinRT apps.

The upshot of all this is that the platform is intentionally designed to provide a particular user experience through WinRT apps. This means that certain types of apps just won't work as WinRT apps, such as file system utilities, antivirus, many kinds of development tools, registry cleaners, and anything else that can't be written with the WinRT APIs (or the available subset of Win32 and .NET APIs; see the next sidebar). In short, if there isn't an available API for the functionality in question, that functionality

isn't supported in the app container. Such apps must presently be written as desktop applications.

Sidebar: Hybrid Apps

WinRT apps written in JavaScript can only access WinRT APIs directly; apps or libraries written in C#, Visual Basic, and C++ also have access to a small subset of Win32 and .NET APIs. (See [Win32 and COM for WinRT apps](#).) Unfair? Not entirely, because you can write a *WinRT component* in those other languages that can the surface functionality built with those other APIs to the JavaScript environment (through the same projection mechanism that WinRT itself uses). Because these components are also compiled into binary dynamic-link libraries (DLLs), they will also typically run faster than the equivalent code written in JavaScript and also offer some degree of intellectual property protection (e.g., hiding algorithms).

Such *hybrid apps*, as they're called, thus use HTML/CSS for their presentation layer and some app logic, and they place the most performance critical or sensitive code in compiled DLLs. The dynamic nature of JavaScript, in fact, makes it a great language for gluing together multiple components. We'll see more in Chapter 16, "WinRT Components."

Different Views of Life: View States and Resolution Scaling

So, the user has tapped on an app tile, the app host has been loaded into memory, and it's ready to get everything up and running. What does the user see?

The first thing that becomes immediately visible is the app's *splash screen*, which is described in its manifest with an image and background color. This system-supplied screen guarantees that at least *something* shows up for the app when it's activated, even if the app completely gags on its first line of code or never gets there at all. In fact, the app has 15 seconds to get its act together and display its main window, or Windows automatically gives it the boot (terminates it, that is) if the user switches away. This avoids having apps that hang during startup and just sit there like a zombie, where often the user can only kill it off by using that most consumer-friendly tool, Task Manager. (Yes, I'm being sarcastic—even though the Windows 8 Task Manager is in fact much more user-friendly.) Of course, some apps will need more time to load, in which case you create an *extended splash screen*. This just means making the initial view of your main window look the same as the splash screen so that you can then overlay progress indicators or other helpful messages like "Go get a snack, friend, 'cause yer gonna be here a while!" Better yet, why not entertain your users so that they have fun with your app even during such a process?

Now, when a normally launched app comes up, it has full command of the entire screen—well, not entirely. Windows reserves a one pixel space along every edge of the display through which it detects edge gestures, but the user doesn't see that detail. Your app still gets to draw in those areas, mind you, but it will not be able to detect pointer events therein. A small sacrifice for full-screen glory!

The purpose of those *edge gestures*—swipes from the edge of the screen toward the center—is to keep both system chrome and app commands (like menus and other commanding UI) out of the way until needed—an aspect of the design principle we call “content before chrome.” This helps keep the user fully immersed in the app experience. To be more specific, the left and right edge gestures are reserved for the system, whereas the top and bottom are for the app. Swiping up from the top or bottom edges, as you’ve probably seen, brings up the *app bar* on the bottom of the screen where an app places most of its commands, and possibly also a *navigation bar* on the top.

When running full-screen, the user’s device can be oriented in either portrait or landscape, and apps can process various events to handle those changes. An app can also specify a preferred *startup orientation* in the manifest and can also *lock* the orientation when appropriate. For example, a movie player will generally want to lock into landscape mode such that rotating the device doesn’t change the display. We’ll see all these layout details in Chapter 6, “Layout.”

What’s also true is that your app might not always be running full-screen. In landscape mode, there are actually three distinct view states that you need to be ready for with every page in the app: *full-screen*, *snapped*, and *filled*. (See Figure 1-6.) These view states allow the user to split the screen into two regions, one that’s 320 pixels wide along either the left or right side of the screen—the *snap region*—and a second that occupies the rest—the *fill region*. In response to user actions, then, your app might be placed in either region and must suck in its gut, so to speak, and adjust its layout appropriately. Most of the time, running in “fill” is almost the same as running in full-screen, except that the display area has slightly different dimensions and a different aspect ratio. Many apps will simply adjust their layout for those dimensions; in some cases, like movies, they’ll just add a letterbox or sidepillar region to preserve the aspect ratio of the content. Both approaches are just fine.

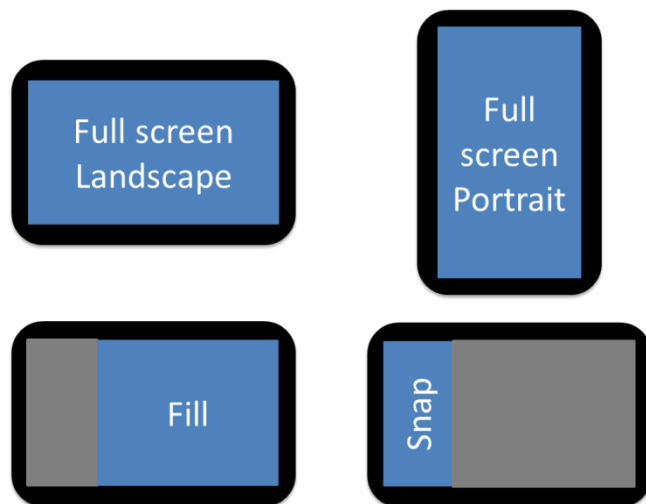


FIGURE 1-6 The four view states for WinRT apps; all pages within the app need to be prepared to show properly in all four view states, a process that generally just involves visibility of elements and layout that can often be handled entirely within CSS media queries.

When snapped, on the other hand, apps will often change the view of their content or its level of detail. Horizontally oriented lists, for instance, are typically switched to a vertical orientation, with fewer details. But don't be nonchalant about this: you really want to consciously design snap views for every page in your app and to design them well. After all, users like to look at things that are useful and beautiful, and the more an app does this with its snap views, the more likely it is that users will keep that app visible even while they're working in another.

Another key point for snapping—and all the view states including portrait—is that they aren't mode changes. The system is just saying something like, "Please stand over here in this doorway, or please lean sideways." So the app should never change what it's doing (like switching from a game board to a high score list) when it's snapped; it should just present itself appropriately for that position. For snap view in particular, if an app can't really continue to run effectively in snap, it should present a message to that effect with an option to un-snap back to full screen. (There's an API for that.)

Beyond the view states, an app should also expect to show itself in many sizes. It will be run on many different displays, anywhere from 1024x768 (the minimum hardware requirement for Windows 8, which also happens to be filled view size on 1366x768), all the way up to resolutions like 2560x1440. The guidance here is that apps with fixed content (like a game board) will generally scale in size across different resolutions, whereas apps with variable content (like a news reader) will generally show more content. For more details, refer to [Designing flexible layouts](#) and [Designing UX for apps](#).

It might also be true that you're running on a high-resolution device that also has a very small screen (high *pixel density*), like 10" screens with a 2560x1440 resolution. Fortunately, Windows does automatic *scaling* such that the app still sees a 1366x768 display through CSS, JavaScript, and the WinRT API. In other words, you almost don't have to care. The only concern is bitmap (raster) graphics, which need to accommodate those scales, as we'll also see in Chapter 6.

As a final note, when an app is activated in response to a contract like Search or Share, its initial view might not be the full window at all but rather its specific landing page for that contract that overlays the current foreground app. We'll see these details in Chapter 12, "Contracts."

Sidebar: Single-Page vs. Multipage Navigation

When you write a web application with HTML, CSS, and JavaScript, you typically end up with a number of different HTML pages and navigate between them by using `<a href>` tags or by setting `document.location`.

This is all well and good and works in a WinRT app, but it has several drawbacks. One is that navigation between pages means reloading script, parsing a new HTML document, and parsing and applying CSS again. Besides obvious performance implication, this makes it difficult to share variables and other data between pages, as you need to either save that data in persistent storage or stringify the data and pass it on the URI.

Furthermore, switching between pages is visually abrupt: the user sees a blank screen while the new page is being loaded. This makes it difficult to provide a smooth, animated transition

between pages as generally seen within the Windows 8 personality—it’s the antithesis of “fast and fluid” and guaranteed to make designers cringe.

To avoid these concerns, WinRT apps written in JavaScript are typically structured as a single HTML page (basically a container `div`) into which different bits of HTML content, called *page controls* in WinJS, are loaded into the DOM at runtime (similar to AJAX). This has the benefit of preserving the script context and allows for transition animations through CSS and/or the WinJS animations library. We’ll see the basics of page loading and navigation in Chapter 3, “App Anatomy and Page Navigation.”

Those Capabilities Again: Getting to Data and Devices

At run time, now, even inside the app container, the app has plenty of room to play and to delight your customers. It can utilize many different controls, as we’ll see in Chapters 4 and 5, styling them however it likes from the prosaic to the outrageous and laying them out on a page according to your designer’s fancies (Chapter 6). It can work with commanding UI like the app bar (Chapter 7) and receive and process *pointer events*, which unify touch, mouse, and stylus as shown in Chapter 9. (With these input methods being unified, you can design for touch and get the others for free; input from the physical and on-screen keyboards are likewise unified.). Apps can also work with *sensors* (Chapter 9), rich media (Chapter 10), animations (Chapter 11), contracts (Chapter 12), *tiles and notifications* (Chapter 13), network communication (Chapter 14), and various devices and printing (Chapter 15). And they can adapt themselves to different regional markets, provide accessibility, and work with various monetization options like advertising, trial versions, and in-app purchases (Chapter 17).

Many of these features and their associated APIs have no implications where user privacy is concerned, so apps have open access to them. These include controls, touch/mouse/stylus input, keyboard input, and sensors (like the accelerometer, inclinometer, and light sensor). The appdata folders (local, roaming, and temp) that were created for the app at installation are also openly accessible. Other features, however, are again under more strict control. As a person who works remotely from home, for example, I really don’t want my webcam turning on unless I specifically tell it to—I may be calling into a meeting before I’ve had a chance to wash up! Such devices and other protected system features, then, are again controlled by a broker layer that will deny access if (a) the capability is not declared in the manifest, or (b) the user specifically disallows that access at run time. Those capabilities are listed in the following table:

Capability	Description	Prompts for user consent at run time
Internet (Client)	Outbound access to the Internet and public networks (which	No

	includes making requests to servers and receiving information in response). ⁶	
Internet (Client & Server) (superset of Internet Client; only one needs to be declared)	Outbound and inbound access to the Internet and public networks (inbound access to critical ports is always blocked).	No
Private Networks (Client & Server)	Outbound and inbound access to home or work intranets (inbound access to critical ports is always blocked).	No
Document Library	Read/write access to the user's Documents area on the file system for specifically declared file types. Requires a corporate account in the Windows Store.	No
Music Library Pictures Library Video Library	Read/write access to the user's Music/Pictures/Videos area on the file system (all files).	No
Removable Storage	Read/write access to files on removable storage devices for specifically declared file types.	No
Microphone	Access to microphone audio feeds (includes microphones on cameras).	Yes
Webcam	Access to camera audio/video/image feeds.	Yes
Location (GPS)	Access to the user's location.	Yes
Proximity	The ability to connect to other devices through near-field communication (NFC).	No
Enterprise Authentication	Access to intranet resources that require domain credentials; not typically needed for most apps. Requires a corporate account in the Windows Store.	No
Shared User Certificates	Access to software and hardware (smart card) certificates. Requires a corporate account in the Windows Store.	Yes, in that the user must take action to select a certificate, insert a smart card, etc.

Note It is also possible for an app to declare access to ad-hoc devices by adding the appropriate hardware class ID to the manifest. See Chapter 15, “Devices and Printing.”

When user consent is involved, calling an API to access the resource in question will prompt for user consent, as shown in Figure 1-7 (from the app we'll create in Chapter 2). If the user accepts, the API call will proceed; if the user declines, the API call will return an error. Apps must accordingly be prepared for such APIs to fail, and they must then behave accordingly.



FIGURE 1-7 A typical user consent dialog that's automatically shown when an app first attempts to use a brokered capability. This will happen only once within an app, but the user can control their choice through the Settings charm for that app.

⁶ Note that network capabilities are not necessary to receive push notifications for “live tiles,” because those are received by the system and not the app.

When you first start writing apps, really keep the manifest and these capabilities in mind—if you forget one, you’ll see APIs failing even though all your code is written perfectly (or was copied from a working sample). In the early days of building the first WinRT apps at Microsoft, we routinely forgot to declare the Internet Client capability, so even things like getting to remote media with an `img` element or making a simple call to a web service would fail. The support for alerting you if you’ve forgotten a capability is much better now, but if you hit some mysterious problem with code that you’re sure should work, especially in the wee hours of the night, check the manifest!

We’ll encounter many other sections of the manifest besides capabilities in this book. For example, the documents library and removable storage capabilities both require you to declare the specific file types for your app (otherwise access will generally be denied). The manifest also contains *content URIs*: specific rules that govern which URIs are known and trusted by your app and can thus act on the app’s behalf. The manifest is also where you declare things like your preferred orientation, *background tasks* (like playing audio or handling real-time communication), contract behaviors (such as which page in your app should be brought up in response to being invoked via a contract), custom protocols, and the appearance of tiles and notifications. Like I said earlier, you and your app become real bosom buddies with the manifest.

The last note to make about capabilities is that while programmatic access to the file system is controlled by certain capabilities, the user can always point your app to other nonsystem areas of the file system—and any type of file—from within the file picker UI. (See Figure 1-8.) This explicit user action, in other words, is taken as consent for your app to access that particular file or folder (depending on what you’re asking for). Once you’re app is given this access, you can use certain APIs to record that permission so that you can get to those files and folders the next time your app is launched.

In summary, the design of the manifest and the brokering layer is to ensure that the user is always in control where anything sensitive is concerned, and as your declared capabilities are listed on your app’s description page in the Windows Store, the user should never be surprised by your app’s behavior.

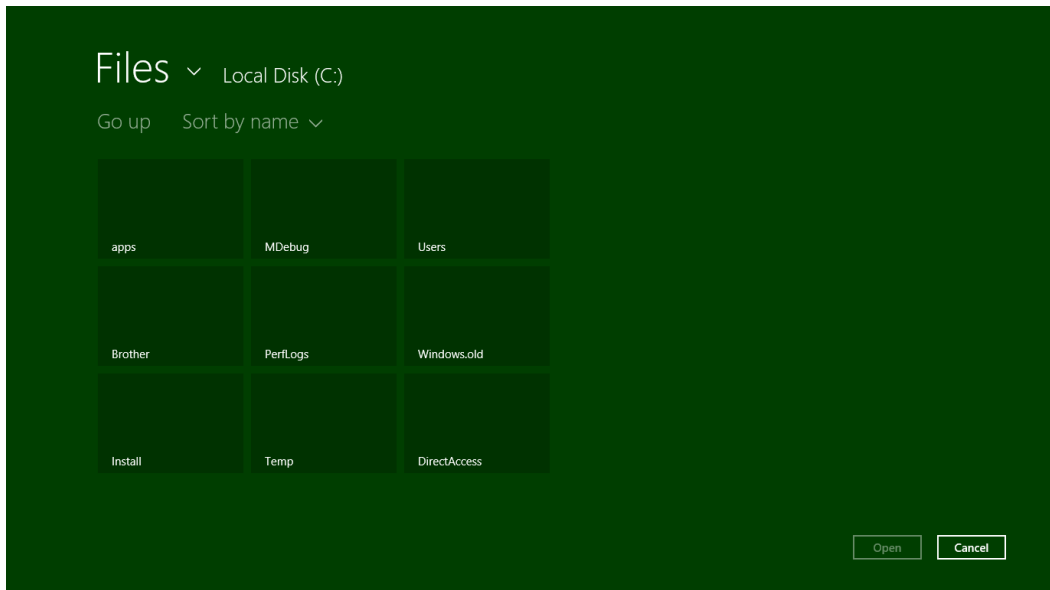


FIGURE 1-8 Using the file picker UI to access other parts of the file system from within a WinRT app, such as folders on a drive root (but not protected system folders). This is done by tapping the down arrow next to “Files.”

Taking a Break, Getting Some Rest: Process Lifecycle Management

Whew! We’ve covered a lot of ground already in this first chapter—our apps have been busy, busy, busy, and we haven’t even started writing any code yet! In fact, apps can become really busy when they implement certain sides of contracts. If an app declares itself as a Search, Share, Contact, or File Picker *source* in its manifest (among other things), Windows will activate the app in response to the appropriate user actions. For example, if the user invokes the Share charm and picks your app as a Share *target*, Windows will activate the app with an indication of that purpose. In response, the app displays its specific share UI or *view*—not the whole app—and when that task is complete, Windows will shut your app down again (or send it to the background if it was already running) without the need for additional user input.

This automatic shutdown or sending the app to the background are examples of automatic *lifecycle management* for WinRT apps that helps conserve power and optimize battery life. One reality of life in traditional multitasking operating systems is that users typically leave a bunch of apps running, all of which consume power. This made sense with desktop apps because many of them can be at least partially visible at once. But for WinRT apps, the system is boldly taking on the job itself and using the full-screen nature of those apps to its advantage.

Apps typically need to be busy and active only when the user can see them (in whatever view state).

So when most apps are no longer visible, there is really little need to keep their engines running on idle. It's better to just turn them off, give them some rest, and let the visible apps utilize the system's resources.

So when an app goes to the background, Windows will automatically *suspend* it after about 5 seconds (according to the wall clock). The app is notified of this event so that it can save whatever state it needs to (which I'll describe more in the next section). At this point the app is still in memory, with all its in-memory structures intact, but it will simply not be scheduled for any CPU time. (See Figure 1-9.) This is very helpful for battery life because most desktop apps idle like a gasoline-powered car, still consuming a little CPU in case there's a need, for instance, to repaint a portion of a window. Because a WinRT app in the background is completely obscured, it doesn't need to do such small bits of work and can be effectively frozen.

If the user then switches back to the app (in whatever view state, through whatever gesture), it will be scheduled for CPU time again and *resume* where it left off (adjusting its layout for the view state, of course). The app is also notified of this event in case it needs to re-sync with online services, update its layout, refresh a view of a file system library, or take a new sensor reading because any amount of time might have passed since it was suspended. Typically, though, an app will not need to reload any of its own state because it was in memory the whole time.

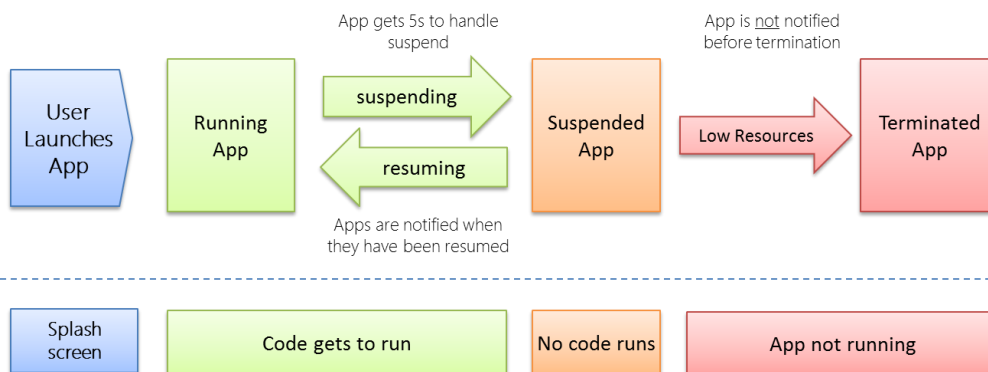


FIGURE 1-9 Process lifetime states for WinRT apps.

There are a couple of exceptions to this. First, Windows provides a *background transfer* API—see Chapter 14, “Networking”—to offload downloads and uploads from app code, which means apps don't have to be running for such transfers to happen. Apps can also ask the system to periodically update *live tiles* on the Start page with data obtained from a service, or they can employ *push notifications* (through the Windows Notification Service, WNS) so that they need not even be running for this purpose—see Chapter 13, “Tiles, Notifications, the Lock Screen, and Background Tasks.” Second, certain kinds of apps that do useful things when they're not visible, such as audio players, communications apps, or those that need to take action when specific system events occur (like a network change, user login, etc.). With audio, as we'll see in Chapter 10, “Media,” an app specifies background audio in its manifest (where else!) and sets certain properties on the appropriate audio elements. With system

events, as we'll also see in Chapter 13, an app declares background tasks in its manifest that are tied to specific functions in their code. In both cases, then, Windows will not suspend the app when it's in the background, or it will wake the app from the suspended state when an appropriate trigger occurs.

Over time, of course, the user might have many WinRT apps in memory, and most of them will be suspended and consume very little power. Eventually there will come a time when the foreground app—especially one that's just been launched—needs more memory than is available. In this case, Windows will automatically *terminate* one or more apps, dumping them from memory. (See Figure 1-9 again.)

But here's the rub: unless a user explicitly closes an app—by using Alt+F4 or a top-to-bottom swipe; Windows Store policy specifically disallows apps with their own close commands or gestures—he or she still rightly thinks that the app is running. So if the user activates it again (as from its tile), the user will expect to return to the same place he or she left off. For example, a game should be in the same place it was before (though automatically paused), a reader should be on the same page, and a video should be paused at the same time. Otherwise, imagine the kinds of ratings and reviews your app will be getting in the Windows Store!

So you might say, “Well, I should just save my app's state when I get terminated, right?” Actually, no: your app will *not* be notified when it's terminated. Why? For one, it's already suspended at that time, so no code will run. In addition, if apps need to be terminated in a low memory condition, the last thing you want is for apps to wake up and try to save state which might require even more memory! It's imperative, as hinted before, that apps save their state when being suspended and ideally even at other checkpoints during normal execution. So let's see how all that works.

Remembering Yourself: App State and Roaming

To step back for a moment, one of the key differences between traditional desktop applications and WinRT apps is that the latter are inherently stateful. That is, once they've run the first time, they remember their state across invocations (unless explicitly closed by the user or unless they provide an affordance to reset the state explicitly). Some desktop applications work like this, but most suffer from a kind of identity crisis when they're launched. Like Gilderoy Lockhart in *Harry Potter and the Chamber of Secrets*, they often start up asking themselves, “Who am I?”⁷ with no sense of where they've been or what they were doing before.

Clearly this isn't a good idea with WinRT apps whose lifetime is being managed automatically. From the user's point of view, apps are always running even if they're not. It's therefore critical that apps save their state when being suspended, in case they get terminated, and that they reload that state if they're

⁷ For those readers who have not watched this movie all the way through the credits, there's a short vignette at the very end. During the movie, Lockhart—a prolific, narcissistic, and generally untruthful autobiographer—loses his memory from a backfiring spell. So in the vignette he's shown in a straitjacket on the cover of his newest book, *Who am I?*

launched again after being terminated. (An app receives a flag on startup to indicate its previous execution state, which determines what it should do with saved state. Details are in Chapter 3.)

There's another dimension to statefulness too. Remember from earlier in this chapter that a user can install the same WinRT app on up to five different devices? Well, that means that an app, depending on its design of course, can also be stateful *between* those devices. That is, if a user pauses a video or a game on one device or has made annotations to a book or magazine on one device, the user will naturally want to be able to go to another device and pick up at exactly the same place.

Fortunately, Windows 8 makes this easy—really easy, in fact—by automatically roaming app settings and state, along with Windows settings, between devices on which the user is logged in with the same Microsoft account, as shown in Figure 1-10.

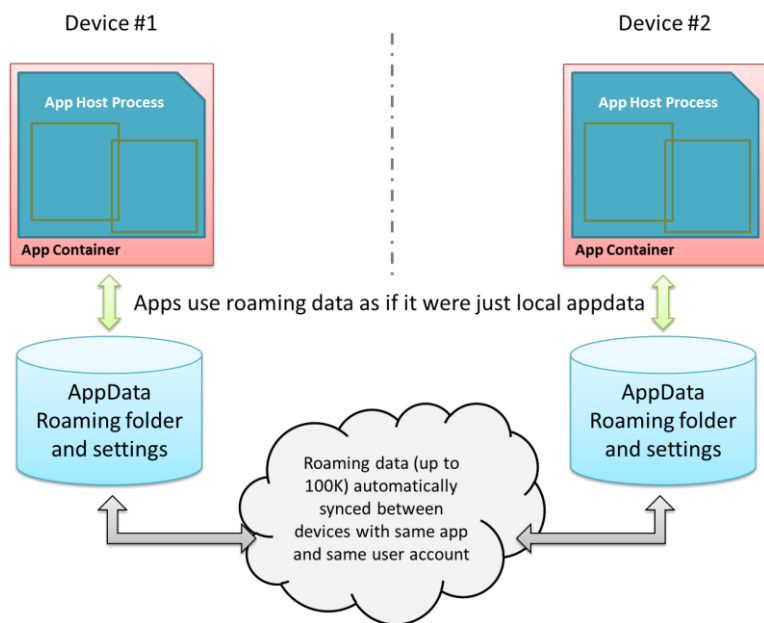


FIGURE 1-10 Automatic roaming of app roaming data (folder contents and settings) between devices.

The key here is understanding how and where an app saves its state. (We already know when.) If you recall, there's one place on the file system where an app has unrestricted access: its appdata folder. Within that folder, Windows automatically creates subfolders named LocalState, RoamingState, and TempState when the app is installed (I typically refer to them without the "State" appended.) The app can programmatically get to any of these folders at any time and can create in them all the files and subfolders to fulfill its heart's desire. There are also APIs for managing individual Local and Roaming settings (key-value pairs), along with groups of settings called *composites* that are always written to, read from, and roamed as a unit. (These are useful when implementing the app's Settings features for the Settings charm, as covered in Chapter 8, "State, Settings, Files, and Documents.")

Now, although the app can write as much as it wants to the app data areas (up to the capacity of the file system), Windows will automatically roam the data in your roaming sections only if you stay below an allowed quota (~100K, but there's an API for that). If you exceed the limit, the data will still be there but none of it will be roamed. Also be aware that cloud storage has different limits on the length of filenames and file paths as well as the complexity of the folder structure. So keep your roaming state small and simple; if the app needs to roam larger amounts of data, use a secondary web service like SkyDrive. (See Chapter 8.)

So the app really needs to decide what kind of state is local to a device and what should be roamed. Generally speaking, any kind of settings, data, or cached resources that are device-specific should always be local (and Temp is also local), whereas settings and data that represent the user's interaction with the app are potential roaming candidates. For example, an email app that maintains a local cache of messages would keep those local but would roam account settings (sans passwords) so that the user has to configure the app on only one device. (It would probably also maintain a per-device setting for how it downloads or updates emails so that the user can minimize network/radio traffic on a mobile device.) A media player, similarly, would keep local caches that are dependent on the specific device's display characteristics, and it would roam playlists, playback positions, favorites, and other such settings (should the user want that behavior, of course).

When state is roamed, know that there's a simple "last writer wins" policy where collisions are concerned. So, if you run the same app on two devices at the same time, don't expect there to be any fancy merging or swapping of state. After all kinds of tests and analysis, Microsoft's engineers finally decided that simplicity was best!

Along these same lines, I'm told that if a user installs an app, roams some settings, uninstalls the app, then within some "reasonable time" reinstalls the app, the user will find that those settings are still in place. This makes sense, because it would be too draconian to blow away roaming state in the cloud the moment a certain user just happened to uninstall an app on all their devices. There's no guarantee of this behavior, mind you, but Windows will apparently retain roaming state for an app for some time at least.

Sidebar: Local vs. Temp Data

For local caching purposes, an app can use either local or temp storage. The difference is that local data will always be under the explicit control of the app. Temp data, on the other hand, can be deleted if the user runs the Disk Cleanup utility. Local data is thus best used to support an app's functionality, and temp data is used to support run-time optimization at the expense of disk space.

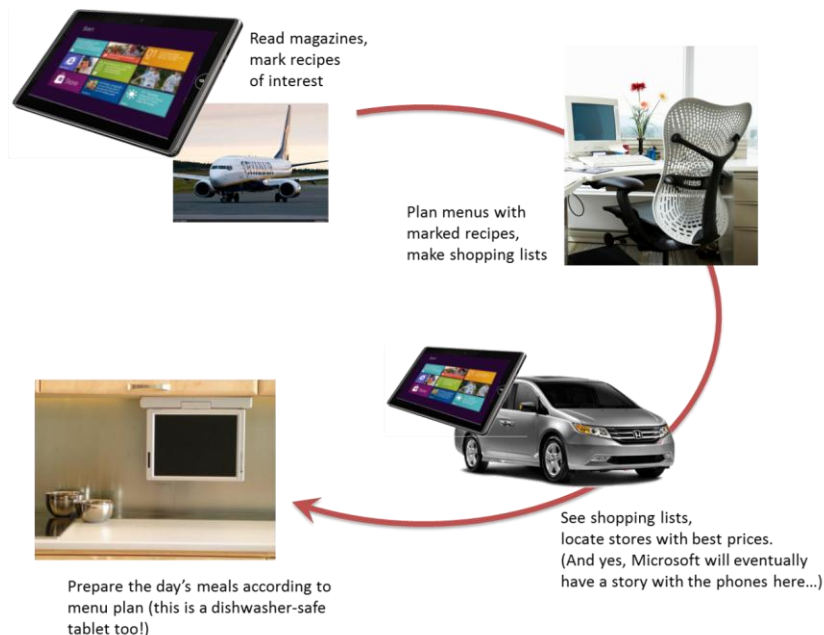
For WinRT apps written in HTML and JavaScript, you can also use existing caching mechanisms like HTML5 local storage, IndexedDB, app cache, and so forth. All of these will be stored within the app's LocalState folder.

Sidebar: The Opportunity of Per-User Licensing and Data Roaming

Details aside, I personally find the cross-device roaming aspect of the platform very exciting, because it enables the developer to think about apps as something beyond a single-device or single-situation experience. As I mentioned earlier, a user's collection of apps is highly personal and it personalizes the device; apps themselves are licensed to the user and not the device. In that way, we as developers can think about each app as something that projects itself appropriately onto whatever device and into whatever context it finds itself. On some devices it can be oriented for intensive data entry or production work, while on others it can be oriented for consumption or sharing. The end result is an overall app experience that is simply more *present* in the user's life and appropriate to each context.

An example scenario is illustrated in Figure 1-11, where an app can have different personalities or flavors depending on user context and how different devices might be used in that context. It might seem rather pedestrian to think about an app for meal planning, recipe management, and shopping lists, but that's something that happens in a large number of households worldwide. Plus it's something that my wife would like to see me implement if I wrote more code than text!

This, to me, is the real manifestation of the next era of personal computing, an era in which personal computing expands well beyond, yet still includes, a single device experience. Devices are merely viewports for your apps and data, each viewport having a distinct role in the larger story of how you move through and interact with the world at large.



Coming Back Home: Updates and New Opportunities

If you're one of those developers that can write a perfect app the first time, I have to ask why you're actually reading this book! Fact of the matter is that no matter how hard we try to test our apps before they go out into the world, our efforts pale in comparison to the kinds of abuse that customers will heap on them. To be more succinct: expect problems. An app might crash under circumstances we never predicted, or there just might be usability problems because people are finding creative ways to use the app outside of its intended purpose.

Fortunately, the Windows Store dashboard—go to <http://dev.windows.com> and click the Dashboard tab at the top—makes it easy for you get the kind of feedback that has traditionally been very difficult to obtain. For one, the Store maintains *ratings and reviews* for every app, which will be a source of valuable insight into how well your app fulfills its purpose in life and a source of ideas for your next release. And you might as well accept it now: you're going to get praise (if you've done a decent job), and you're going to get criticism, even a good dose of nastiness (even if you've done a decent job!). Don't take it personally—see every critique as an opportunity to improve, and be grateful that people took the time to give feedback. As a wise man once said upon hearing of the death of his most vocal critic, "I've just lost my best friend!"

The Store will also provide you with crash *analytics* so that you can specifically identify problem areas in your app that evaded your own testing. This is incredibly valuable—if you're not already clapping your hands in delight!—because if you've ever wanted this kind of data before, you've had to implement the entire mechanism yourself. No longer. This is one of the valuable services you get in exchange for your annual registration with the Store. (Of course, you can still implement your own too.)

With this data in hand and all the other ideas you either had to postpone from your first release or dreamt up in the meantime, you're all set to have your app come home for some new love before its next incarnation.

Updates are onboarded to the Windows Store just like the app's first version. You create and upload an app package (with the same package name as before but a new version number), and then you update your description, graphics, pricing, and other information. After that your updated package goes through the same certification and signing process as before, and when all that's complete your new app will be available in the Store. Those customers who already have your app will also be notified that there's an update, which they can choose to install or not. (And remember that with the blockmap business described earlier, only those parts of the app that have actually changed will be downloaded for an update. This means that issuing small fixes won't force users to repeat potentially large downloads each time, bringing the update model closer to that of web applications.)

When a user installs an update that has the same package name as an existing app, note that all the settings and appdata for the prior version remain intact. Your updated app should be prepared, then, to migrate a previous version of its state if and when it encounters such.

This brings up an interesting question: what happens with roaming data when a user has different versions of the same app installed on multiple devices? The answer is twofold: first, roaming data has its own version number independent of the app, and second, Windows will transparently maintain multiple versions of the roaming state so long as there are apps installed on the user's devices that reference those state versions. Once all the devices have updated apps and have converted their state, Windows will delete old versions.

Another interesting question with updates is whether you can get a list of the customers who have acquired your app from the Store. The answer is no, because of privacy considerations. However, there is nothing wrong with including a registration feature in your app through which users can opt in to receive additional information from you, such as more detailed update notifications. Your Settings panel is a great place to include this.

The last thing to say about the Store is that in addition to analytics about your own app—which also includes data like sales figures, of course—it also provides you with marketwide analytics. These help you explore new opportunities to pursue—maybe taking an idea you had for a feature in one app and breaking that out into a new app in a different category. Here you can see what's selling well (and what's not) or where a particular category of app is underpopulated or generally has less than average reviews. For more details, again see the Dashboard at <http://dev.windows.com>.

And, Oh Yes, Then There's Design

In this first chapter we've covered the nature of the world in which WinRT apps live and operate. In this book, too, we'll be focusing on the details of how to build such apps with HTML, CSS, and JavaScript. But what we haven't talked about, and what we'll only be treating minimally, is how you decide what your app does—its purpose in the world!—and how it clothes itself for that purpose.

This is really the question of good design for WinRT apps—all the work that goes into apps before we even start writing code.

I said that we'll be treating this minimally because I simply do not consider myself a designer. I encourage you to be honest about this yourself: if you don't have a good designer working with you, **get one**. Sure, you can probably work out an OK design on your own, but the demands of a consumer-oriented market combined with a newer design language like that employed in Windows 8—where the emphasis is on simplicity and tailored experiences—underscores the need for professional help. It'll make the difference between a functional app and a great app, between a tool and a piece of art, between apps that consumers accept and those they *love*.

With design, I do encourage developers to peruse the material on [Designing UX for apps](#) for a better understanding of design principles. But let's be honest: as a developer, do you really want to ponder what "fast and fluid" means? Do you want to spend your time in graphic design and artwork (which is essential for a great app)? Do you want to haggle over the exact pixel alignment of your layout in all four view states? If not, find someone who does, because the combination of their design

sensibilities and your highly productive hacking will produce much better results than either of you working alone. As one of my co-workers puts it, a marriage of “freaks” and “geeks” often produces the most creative, attractive, and inspiring results.

Let me add that design is neither a one-time nor a static process. Developers and designers will need to work together throughout the development experience, as design needs will arise in response to how well the implementation really works. For example, the real-world performance of an app might require the use of progress indicators when loading certain pages or might be better solved with a redesign of page navigation. It may also turn out, as we found with one of our early app partners, that the kinds of graphics called for in the design simply weren’t available from the app’s back-end service. The design was lovely, in other words, but couldn’t actually be implemented, so a design change was necessary. So make sure that your ongoing relationship with your designers is a healthy and happy one. And on that note, let’s get into your part of the story: the coding!

Chapter 2

Quickstart

This is a book about developing apps. So, to quote Paul Bettany's portrayal of Geoffrey Chaucer in *A Knight's Tale*, "without further gilding the lily, and with no more ado," let's create some!

A Really Quick Quickstart: The Blank App Template

We must begin, of course, by paying due homage to the quintessential "Hello World" app, which we can achieve without actually writing any code at all. We simply need to create a new app from a template in Visual Studio:

1. Run Visual Studio Express. If this is your first time, you'll be prompted to obtain a developer license. Do this, because you can't go any further without it!
2. Click New Project... in the Visual Studio window.
3. In the dialog that appears (Figure 2-1), make sure you select JavaScript under Templates on the left side, and then select Blank Application in the middle. Give it a name (**HelloWorld** will do), a folder, and click OK.

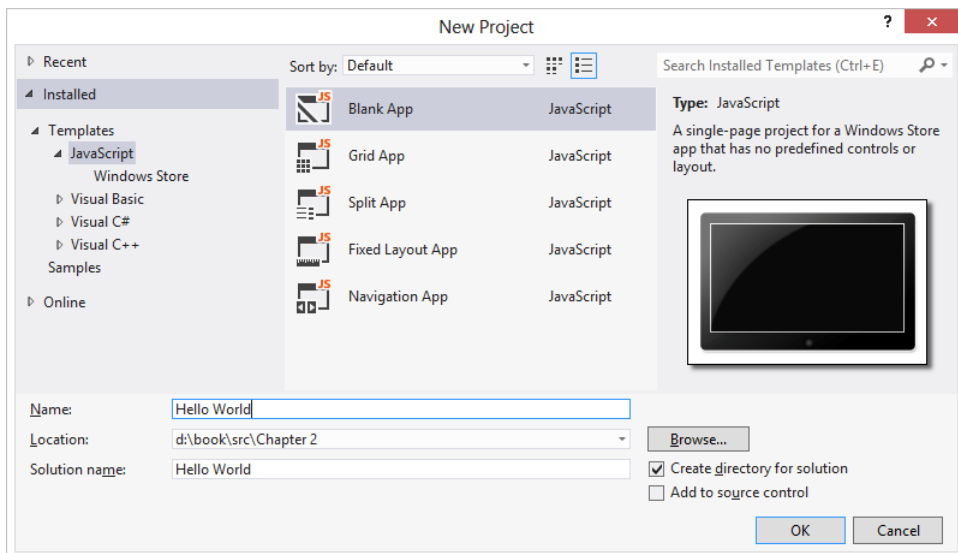


FIGURE 2-1 Visual Studio's New Project dialog using the light UI theme. (See the Tools > Options menu command, and then change the theme in the Environment/General section).

4. After Visual Studio churns for a bit to create the project, click the Start Debugging button (or press F5, or select the same command from the Debug menu). Assuming your installation is good, you should see something like Figure 2-2 on your screen.

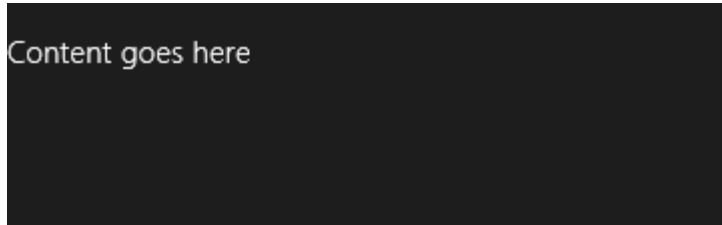


FIGURE 2-2 The only vaguely interesting portion of the Hello World app's display. The message is at least a better invitation to write more code than the standard first-app greeting!

By default, Visual Studio starts the debugger in *local machine* mode, which runs the app full screen on your present system. This has the unfortunate result of hiding the debugger unless you're on a multimonitor system, in which case you can run Visual Studio on one monitor and your WinRT app on the other. Very handy. See <http://msdn.microsoft.com/en-us/library/windows/apps/hh441483.aspx> for more on this.

Visual Studio also offers two other debugging modes available from the drop-down list on the toolbar (Figure 2-3) or the Debug/[Appname] Properties menu command (Figure 2-4):

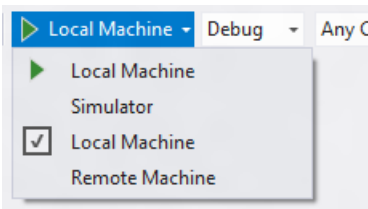


FIGURE 2-3 Visual Studio's debugging options on the toolbar.

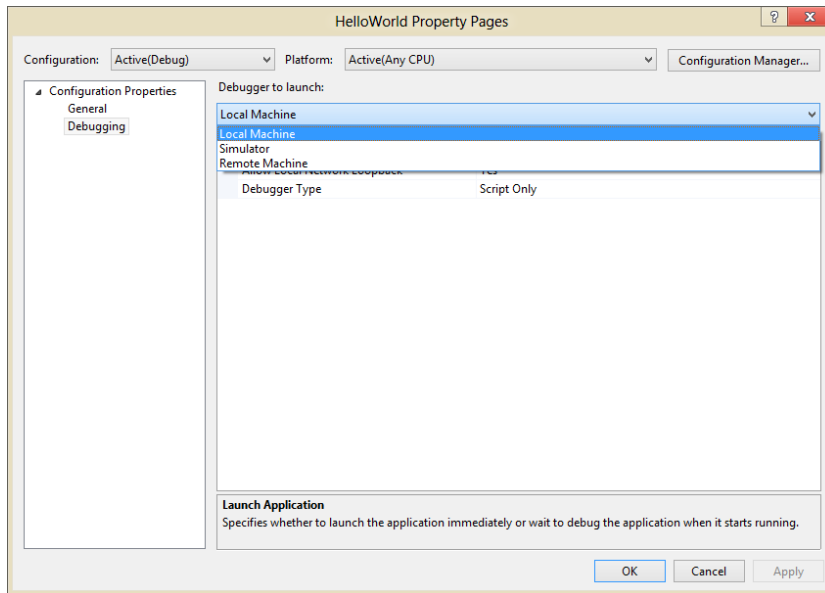


FIGURE 2-4 Visual Studio’s debugging options in the app properties dialog.

The Remote Machine option allows you to run the app on a separate device, which is absolutely essential for working with devices that can’t run desktop apps at all. We won’t cover this topic in this book, but it’s straightforward; see [Running WinRT apps on a remote machine](#) for details. Also, when you don’t have a project loaded in Visual Studio, the Debug menu offers the Attach To Process command, which allows you to debug an already-running app. For this I defer once more to the documentation: [How to start a debugging session \(JavaScript\)](#).

The Simulator is also very interesting, really the most interesting option in my mind and a place I imagine you’ll be spending plenty of time. It duplicates your environment inside a new login session and allows you to control device orientation, set various screen resolutions (and scaling factors), simulate touch events, and control the data returned by geolocation APIs. Figure 2-5 shows Hello World in the simulator with the additional controls labeled. We’ll see more of the simulator as we go along, though you may also want to peruse the [Running WinRT apps in the simulator](#) topic.

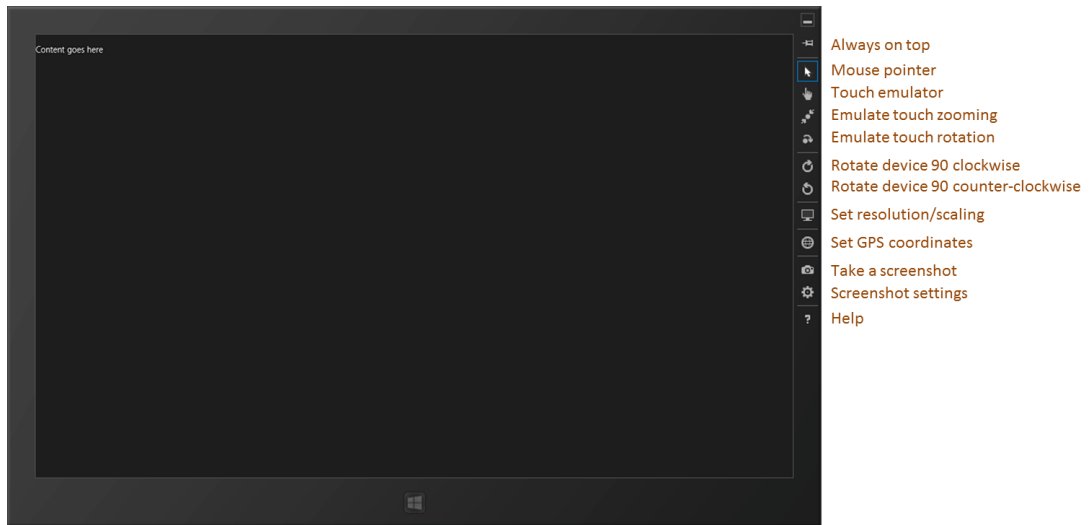


FIGURE 2-5 Hello World running in the simulator, with labels on the right for the simulator controls. Truly, the “Blank App” template lives up to its name!

Sidebar: How Does Visual Studio Run an App?

Under the covers, Visual Studio is actually deploying the app similar to what would happen if you acquired it from the Store. The app will show up on the Start page, where you can also uninstall it (a helpful step if you want to reset the appdata folders and other state).

There’s really no magic involved: deployment can actually be done through the command line. To see the details, use the Store/Create App Package in Visual Studio, select No for a Store upload, and you’ll see a dialog in which you can save your package wherever you want. In that folder you’ll then find an appx package, a security certificate, and a batch file called *Add-AppxDevPackage*. That batch file contains PowerShell scripts that will deploy the app along with its dependencies.

These same files are also what you can share with other developers to side-load your app without having to give them your source project.

Blank App Project Structure

While an app created with the Blank template doesn’t have much in the visual department, it provides much more where project structure is concerned. Here’s what you’ll find coming from the template, which is found in Visual Studio’s Solution Explorer (as shown in Figure 2-6):

In the project root folder:

- **default.html** The starting page for the app.

- **package.appmanifest** The manifest. Opening this file will show Visual Studio's manifest editor (shown later in this chapter). I encourage you to browse around in this UI for a few minutes to familiarize yourself with what's all here. For example, you'll see references to the images noted below, a checkmark on the Internet Client capability checked, default.html selected as the start page, and all the places where you control different aspects of your app. We'll be seeing these throughout this book; for a complete reference, see the [App packages and deployment](#) and [Manifest designer](#) topics. And if you want to explore the manifest XML directly, right-click this file and select View Code.
- **<Appname>_TemporaryKey.pfx** A temporary signature created on first run.

The css folder contains a core default.css file where you'll see media query structures for the four view states that all apps should honor. We'll see this in action in the next section, and I'll discuss all the details in Chapter 6, "Layout."

The images folder contains four reference images, and unless you want to look like a real doofus developer, you'll *always* want to customize these before your app is complete (along with providing scaled versions as we'll see in Chapter 3, "App Anatomy and Page Navigation"):

- **logo.png** A default 150x150 (100% scale) image for the Start page.
- **smalllogo.png** A 30x30 image for the zoomed-out Start page.
- **splashscreen.png** A 620x300 image that will be shown while the app is loading.
- **storelogo.png** A 50x50 image that will be shown for the app in the Windows Store. This needs to be part of an app package but is not used within Windows at run time.

The js folder contains a simple default.js.

The References folder points to CSS and JS files for the WinJS library. You can open any of these to see how WinJS itself is implemented. (Note: if you want to search within these files, you must open and search only within the specific file. These are not included in solution-wide or project-wide searches.)

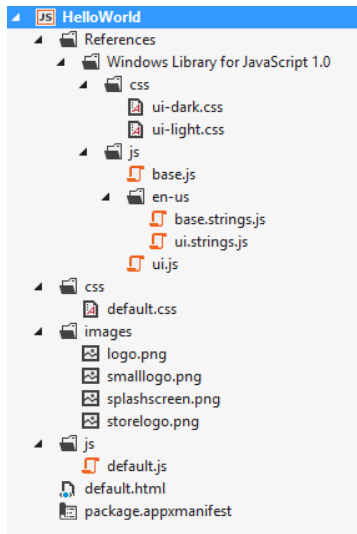


FIGURE 2-6 A Blank app project fully expanded in Solution Explorer.

As you would expect, there's not much app-specific code for this type of project. For example, the HTML has only a single paragraph element in the body, the one you can replace with "Hello World" if you're really not feeling complete without doing so. What's more important at present are the references to the WinJS components: a core stylesheet (ui-dark.css or ui-light.css), base.js, and ui.js:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Hello World</title>

  <!-- WinJS references -->
  <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet">
  <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
  <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

  <!-- HelloWorld references -->
  <link href="/css/default.css" rel="stylesheet">
  <script src="/js/default.js"></script>
</head>
<body>
  <p>Content goes here</p>
</body>
</html>
```

You will generally always have these references (perhaps using `ui-light.css` instead) in every HTML file of your project. The `//`'s in the WinJS paths refer to shared libraries rather than files in your app package, whereas a single `/` refers to the root of your package. Beyond that, everything else is standard HTML5, so feel free to play around with adding some additional HTML of your own and see the effect.

Where the JavaScript is concerned, `default.js` just contains the basic WinJS activation code centered on the `WinJS.Application.onactivated` event along with a stub for an event called `WinJS.Application.oncheckpoint`:

```
(function () {
    "use strict";

    var app = WinJS.Application;
    var activation = Windows.ApplicationModel.Activation;
    WinJS.strictProcessing();

    app.onactivated = function (args) {
        if (args.detail.kind === activation.ActivationKind.launch) {
            if (args.detail.previousExecutionState !==
                activation.ApplicationExecutionState.terminated) {
                // TODO: This application has been newly launched. Initialize
                // your application here.
            } else {
                // TODO: This application has been reactivated from suspension.
                // Restore application state here.
            }
            args.setPromise(WinJS.UI.processAll());
        }
    };

    app.oncheckpoint = function (args) {
    };

    app.start();
})();
```

We'll come back to `checkpoint` in Chapter 3 and `WinJS.strictProcessing` in Chapter 4, "Controls, Control Styling, and Data Binding." For now, remember from Chapter 1, "The Life Story of a WinRT App," that an app can be activated in many ways. These are indicated in the `args.detail.kind` property, whose values come from the `Windows.ApplicationModel.Activation.ActivationKind` enumeration.

When an app is launched directly from its tile on the Start screen (or in the debugger as we've been doing), the kind is just `launch`. As we'll see later on, other values tell us when an app is activated to service requests like the search or share contracts, file-type associations, file pickers, protocols, and more. For the `launch` kind, another bit of information from the `Windows.ApplicationMode.Activation.ApplicationExecutionState` enumeration tells the app how it was last running. Again, we'll see more on this in Chapter 3, so the comments in the default code above should satisfy your curiosity for the time being.

Now, what is that `args.setPromise(WinJS.UI.processAll())` for? As we'll see many times, `WinJS.UI.processAll` instantiates any WinJS controls that are declared in HTML—any element (commonly a `div` or `span`) that contains a `data-win-control` attribute whose value is the name of a

constructor function. Of course, the Blank app template doesn't include any such controls, but because just about every app based on this template *will*, it makes sense to include it by default.⁸ As for `args.setPromise`, that's employing something called a deferral that we'll fittingly defer to Chapter 3.

That little `app.start();` at the bottom is also a very important piece, even for as short as it is. It makes sure that various events that were queued during startup get processed. We'll again see the details in Chapter 3.

Finally, you may be asking, "What on earth is all that ceremonial `(function () { ... })();` business about?" It's just a conventional way in JavaScript (called the *module pattern*) to keep the global namespace from becoming polluted, thereby propitiating the performance gods. The syntax defines an anonymous function that's immediately executed, which creates a function scope for everything inside it. So variables like `app` along with all the function names are accessible throughout the module but don't appear in the global namespace.⁹

You can still introduce variables into the global namespace, of course, and to keep it all organized, WinJS offers a means to define your own namespaces and classes (see `WinJS.Namespace.define` and `WinJS.Class.define`), again helping to minimize additions to the global namespace.

Now that we've seen the basic structure of an app, let's build something more functional and get a taste of the WinRT APIs and a few other platform features.

Sidebar: Writing Code in Debug Mode

Because of the dynamic nature of JavaScript, it's impressive that the Visual Studio team figured out how to make the IntelliSense feature work quite well in the Visual Studio editor. (If you're unfamiliar with IntelliSense, it's the productivity service that provides auto-completion for code as well as popping up API reference material directly inline; learn more at [JavaScript IntelliSense](#).) That said, a helpful trick to make IntelliSense work even better is to write code while Visual Studio is in debug mode. That is, set a breakpoint at an appropriate place in your code, and then run the app in the debugger. When you hit that breakpoint, you can then start writing and editing code, and because the script context is fully loaded, IntelliSense will be working against instantiated variables and not just what it can derive from the source code by itself. You can also use Visual Studio's Immediate pane to execute code directly to see the results.

⁸ There is a similar function `WinJS.Binding.processAll` that processes `data-win-bind` attributes (Chapter 4), and `WinJS.Resources.processAll` that does resource lookup on `data-win-res` attributes (Chapter 17).

⁹ See Chapter 2 of Nicolas Zakas's *High Performance JavaScript* (O'Reilly, 2010) for the performance implications scoping.

QuickStart #1: Here My Am! and an Introduction to Blend for Visual Studio

When my son was three years old, he never—despite the fact that he was born to two engineers parents and two engineer grandfathers—peeked around corners or appeared in a room saying “Hello world!” No, his particular phrase was “Here my am!” Using that particular variation of announcing oneself to the universe, this next app can capture an image from a camera, locate your position on a map, and share that information through the Windows 8 Share charm. Does this sound complicated? Fortunately, the WinRT APIs actually make it quite straightforward!

Sidebar: How Long Did It Take to Write This App?

This app took me about three hours to write. “Oh sure,” you’re thinking, “you’ve already written a bunch of apps, so it was easy for you!” Well, yes and no. For one thing, I also wrote this part of the chapter at the same time, and endeavored to make some reusable code. But more importantly, it took a short amount of time because I learned how to use my tools—especially Blend—and I knew where I could find code that already did most of what I wanted, namely all the Windows SDK samples that you can download from <http://code.msdn.microsoft.com/windowsapps/>.

As we’ll be drawing from many of these most excellent samples in this book, I encourage you to go download the whole set—go to the URL above, and the first download below the featured ones will take you to a page where you can get a .zip file with all the JavaScript samples. Once you unzip these, get into the habit of searching that folder for any API or feature you’re interested in. For example, the code I use below to implement camera capture and sourcing data via share came directly from a couple of samples. (By the way, if you open a sample that seems to support only the Remote Machine debugging option, the build target is probably set to ARM—change it to x86 or x64 for local debugging.)

I also *strongly* encourage you to spend a day, even a half-day, getting familiar with Visual Studio and Blend for Visual Studio and just perusing through the samples so that you know what’s there. Trust me: such small investments will pay huge productivity dividends even in the short term!

Design Wireframes

Before we start on the code, let’s first look at design wireframes for this app. Oooh...design? Yes! Perhaps for the first time in the history of Windows, there’s a real design *philosophy* to apply to apps. In the past, with desktop apps, it’s been more of an “anything goes” scene. There were some UI guidelines, sure, but developers could generally get away with making up whatever user experience that made sense to them, like burying essential checkbox options four levels deep in a series of modal

dialog boxes. Yes, this kind of stuff does make sense to certain kinds of developers; whether it makes sense to anyone else is highly questionable!

If you've ever pretended or contemplated pretending to be a designer, now is the time to surrender that hat to someone with real training or set development aside for a year or two and invest in that training yourself. Simply said, *design matters* for WinRT apps, and it will make the difference between apps that merely exist in the Windows Store and are largely ignored and apps that succeed. And having a design in hand will just make it easier to implement because you won't have to make those decisions when you're writing code! (If you still intend on filling designer shoes and communing with apps like Adobe Illustrator, be sure to visit <http://design.windows.com> for the philosophy and details of WinRT app design, plus design resources.)

When I had the idea for this app, I drew up a simple wireframe, let a few designers laugh at me behind my back, and landed on layouts for the full screen, portrait, snap, and fill view states as shown in Figure 2-7 and Figure 2-8.

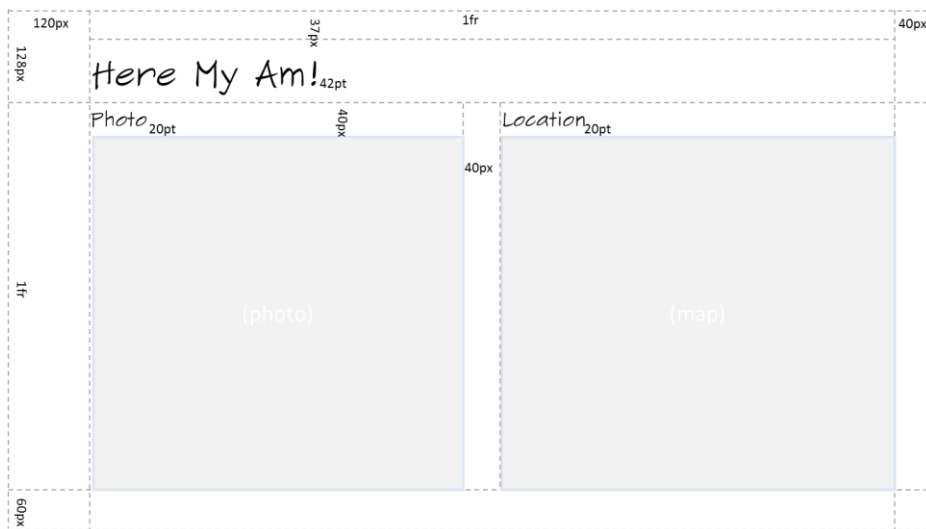


FIGURE 2-7 Full-screen landscape and filled (landscape) wireframe. These states typically use the same wireframe (the same margins), with the proportional parts of the grid simply becoming smaller with the reduced width.

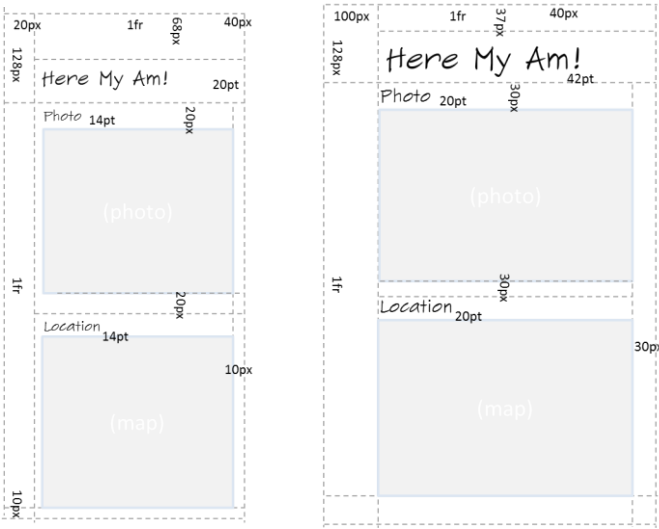


FIGURE 2-8 Snapped wireframe (left; landscape only) and full-screen portrait wireframe (right).

Sidebar: Design for All Four View States!

Just as I thought about all four view states together for Here My Am!, I encourage you to do the same for one simple reason: *your app will be put into every view state whether you design for it or not*. Users, not the app, control the view states, so if you neglect to design for any given state, your app will probably look hideous in that state. You can, as we'll see in Chapter 6, lock the landscape/portrait orientation for your app if you want, but that's meant to enhance an app's experience rather than being an excuse for indolence. So in the end, unless you have a very specific reason not to, every page in your app needs to anticipate all four view states.

This might sound like a burden, but view states don't affect function: they are simply different views of the same information. Remember that changing the view state never changes the *mode* of the app. Handling the view states, therefore, is primarily a matter of which elements are visible and how those elements are laid out on the page. It doesn't have to be any more complicated than that.

One of the important aspects of WinRT app design is following what's called the layout *silhouette*: the size of the header fonts, their placement, the specific margins, and all that (as marked in the previous figures). It might seem restrictive, but the purpose of this recommendation is to encourage a high degree of consistency between apps so that users' eyes literally develop muscle memory for common elements of the UI. Some of this can be found in [Understanding the Windows 8 silhouette](#) and is otherwise incorporated into the templates along with many other design aspects. It's one reason why Microsoft generally recommends starting new apps with a template and going from there. What I show in the wireframes above reflects the layouts provided by one of the more complex templates.

Enough said! Let's just assume that we have a great design to work from and our designers are off sipping cappuccino, satisfied with a job well done. Our job is how to then execute on that great design.

Create the Markup

For the purposes of markup, layout, and styling, one of the most powerful tools you can add to your arsenal is Blend for Visual Studio. As you may know, Blend has been available (at a high price) to designers and developers working with XAML (the presentation framework that is used by WinRT apps written in C#, Visual Basic, and C++). Now Blend is free and also supports HTML, CSS, *and* JavaScript. I emphasize that latter point because it doesn't just load markup and styles: it loads and *executes* your code, right in the "Artboard" (the design surface), because that code so often affects the DOM, styling, and so forth. Then there's Interactive Mode...but I'm getting ahead of myself!

Blend and Visual Studio are very much two sides of a coin: they share the same project file formats and have commands to easily switch between them, depending on whether you're focusing on design or development. To demonstrate that, let's actually start building Here My Am! in Blend. As we did before with Visual Studio, launch Blend, select New Project..., and select the Blank App template. This will create the same project structure as below. (Note: Video 2-1 shows all these steps together.)

Following the practice of writing pure markup in HTML—with no styling and no code, and even leaving off a few classes we'll need for styling—let's drop the following markup into the `body` element of `default.html` (replacing the one line of `<p>Content goes here</p>`):

```
<div id="mainContent">
  <header aria-label="Header content" role="banner">
    <h1 class="titlearea win-type-ellipsis">
      <span class="pagetitle">Here My Am!</span>
    </h1>
  </header>
  <section aria-label="Main content" role="main">
    <div id="photoSection" aria-label="Photo section">
      <h2 class="group-title" role="heading">Photo</h2>
      
    </div>
    <div id="locationSection" aria-label="Location section">
      <h2 class="group-title" role="heading">Location</h2>
      <iframe id="map" src="ms-appx-web:///html/map.html" aria-label="Map"></iframe>
    </div>
  </section>
</div>
```

Here we see the five elements in the wireframe: a main header, two subheaders, a space for a photo (defaulting to an image with "tap here" instructions), and an `iframe` that specifically houses a page in

which we'll instantiate a Bing maps web control.¹⁰

You'll see that some elements have style classes assigned to them. Those that start with `win-` come from the WinJS stylesheet.¹¹ You can browse these in Blend by using the Style Rules tab, shown in Figure 2-9. Other styles like `titlearea`, `pagetitle`, and `group-title` are meant for you to define in your own stylesheet, thereby overriding the WinJS styles for particular elements.

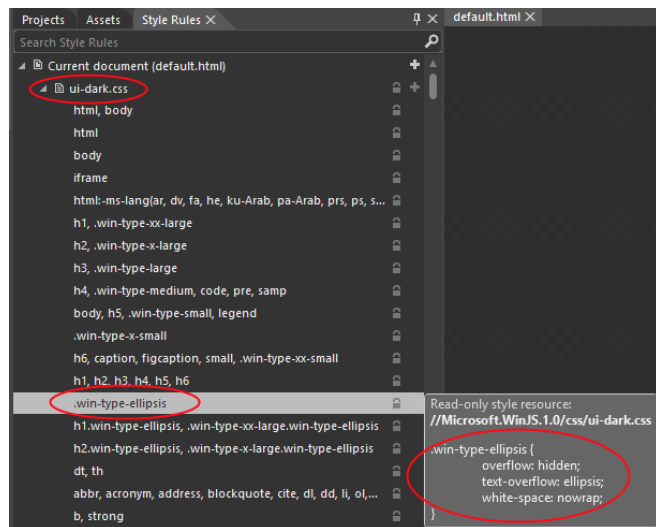


FIGURE 2-9 In Blend, the Style Rules tab lets you look into the WinJS stylesheet and see what each particular style contains. Take special notice of the search bar under the tabs. This is here so you don't waste your time visually scanning for a particular style—just start typing in the box, and let the computer do the work!

The page we'll load into the `iframe`, `map.html`, is part of our app package that we'll add in a moment, but note how we reference it. The `ms-appx-web:///` protocol indicates that the `iframe` and everything inside it will run in the web context (introduced in Chapter 1), thereby allowing us to load the remote script for the Bing maps control. The *triple slash*, for its part—or more accurately the third slash—is shorthand for “the current app package” (a value that you can obtain from `document.location.host`), so we don't need to create an absolute URI.

To indicate that a page should be loaded in the local context, the protocol is just `ms-appx:///`. It's important to remember that no script (including variables and functions) is shared between these contexts; communication between the two goes through the HTML5 `postMessage` function, as we'll see later.

¹⁰ If you're following the steps in Blend yourself, the `taphere.png` image should be added to the project in the `images` folder. Right-click that folder, select `Add Existing Item`, and then navigate to the complete sample's `images` folder and select `taphere.png`. That will copy it into your current project.

¹¹ The two standard stylesheets are `ui-dark.css` and `ui-light.css`. Dark styles are recommended for apps that deal with media, where a dark background helps bring out the graphical elements. We'll use this stylesheet because we're doing photo capture. The light stylesheet is recommended for apps that work more with textual content.

I've also included various `aria-*` attributes on these elements (as the templates do) that support accessibility. We'll look at accessibility in detail in Chapter 17, "Apps for Everyone," but it's an important enough consideration that we should be conscious of it from the start: a majority of Windows users use accessibility features in some way. And although some aspects of accessibility are easy to add later on, adding `aria-*` attributes in markup is best done early.

Styling in Blend

At this point, and assuming you were paying enough attention to read the footnotes, Blend's real-time display of the app shows an obvious need for styling, just like raw markup should. See Figure 2-10.

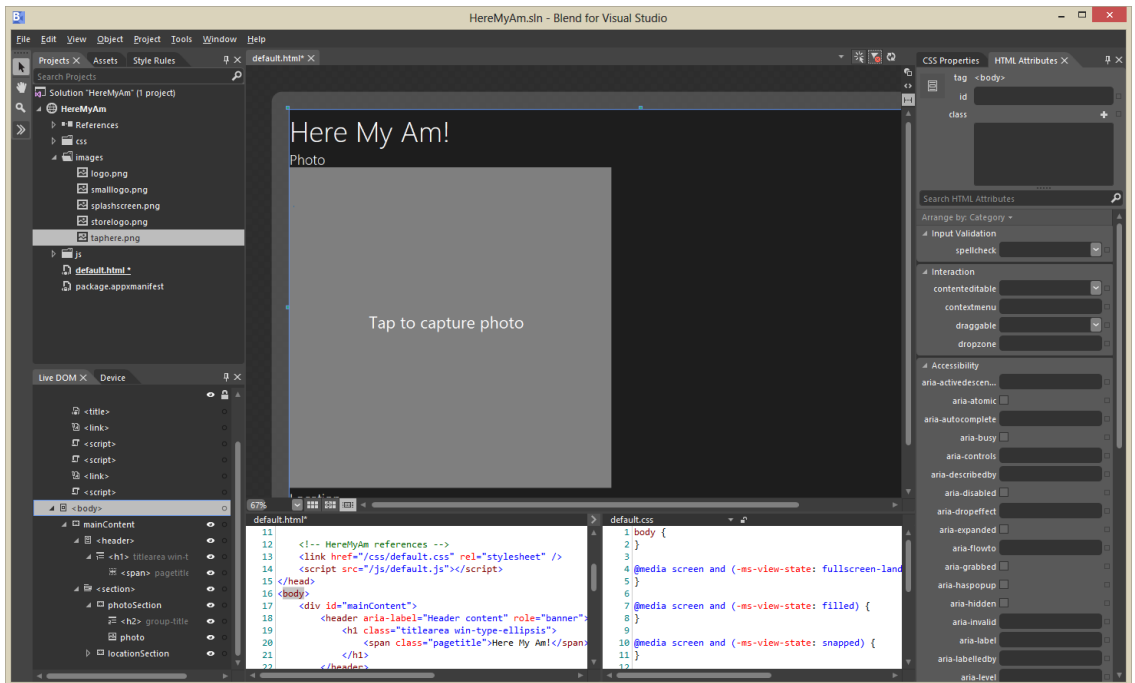


FIGURE 2-10 The app in Blend without styling, showing a view that is much like the Visual Studio simulator. If the `taphere.png` image doesn't show after adding it, use the View/Refresh menu command.

The tabs along the upper left in Blend give you access to your Project files; Assets like all the controls you can add to your UI; and a browser for all the Style Rules defined in the environment. On the lower left side, the Live DOM area lets you browse your element hierarchy and the Device tabs lets you set orientation, screen resolution, and view state;. Clicking an element in the Live DOM here will highlight it in the designer, just like clicking an element in the designer will highlight it in the Live DOM section.

Over on the right side you see what will become a very good friend: the section for HTML Attributes and CSS Properties. In the latter case, the list at the top shows all the sources for styles that are being

applied to the currently selected element and where exactly those styles are coming from (often a headache with CSS). What's selected in that box, mind you, will determine where changes in the properties pane below will be written, so be very conscious of your selection!

Now to get our gauche, unstylish page to look like the wireframe, we need to go through the elements and create the necessary selectors and styles. First, I recommend creating a 1x1 grid in the `body` element as this makes Blend's display in the artboard work better at present. So add `display: -ms-grid; -ms-grid-rows: 1fr; -ms-grid-columns: 1fr;` to `default.css` for that element.

CSS grids also make this app's layout fairly simple: we'll just use a couple of nested grids to place the main sections and the subsections within them, following the general pattern of styling that works best in Blend:

- Right-click the element you want to style in the Live DOM, and select Create Style Rule From Element Id or Create Style Rule From Element Class.

Note If both of these items are disabled, go to the HTML Attributes pane (upper right) and add an id, class, or both. Otherwise you'll be hand-editing the stylesheets later on to move styles around, so you might as well save yourself the trouble.

This will create a new style rule in the app's stylesheet (e.g., `default.css`). In the CSS properties pane on the right, then, find the rule that was created and add the necessary style properties in the pane below.

- Repeat with every other element.

If you look in the `default.css` file, you'll notice that the `body` element is styled with a 1x1 grid—leave this in place, because it makes sure the rest of your styling adapts to the screen size.

So for the `mainContent` `div`, we create a rule from the Id and set it up with `display: -ms-grid; -ms-grid-columns: 1fr;` and `-ms-grid-rows: 128px 1fr 60px;`. (See Figure 2-11.) This creates the basic vertical areas for the wireframes. In general, you won't want to put left or right margins directly in this grid because the lower section will often have horizontally scrolling content that should bleed off the left and right edges. In our case we could use one grid, but instead we'll add those margins in a nested grid within the header and section elements.

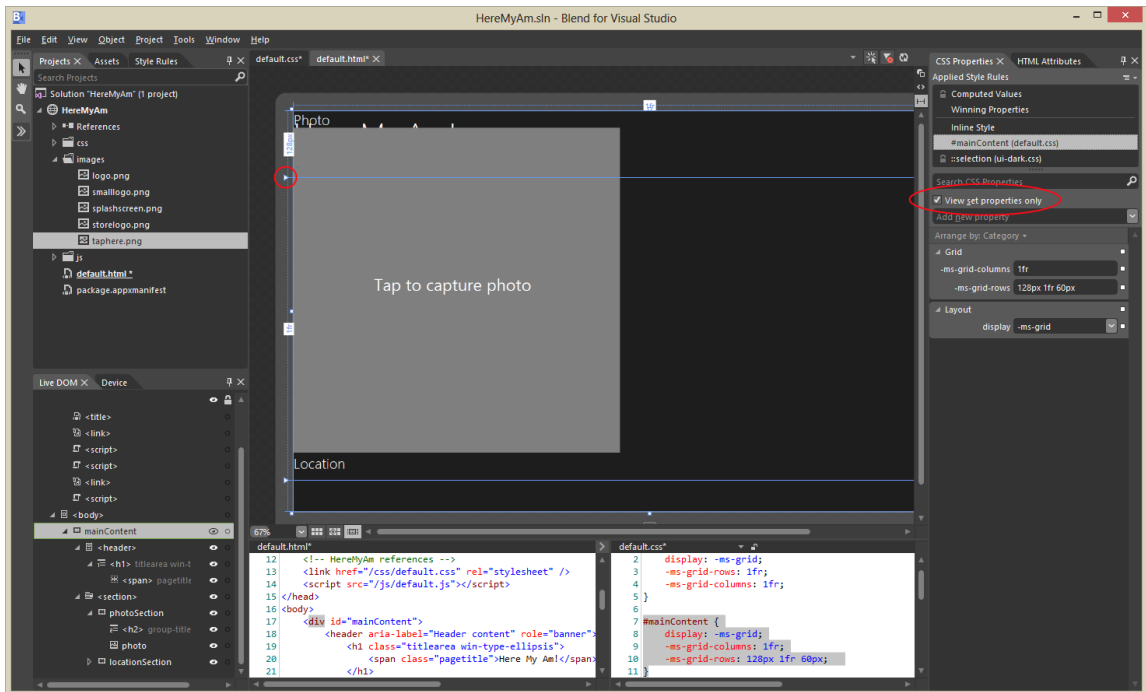


FIGURE 2-11 Setting the grid properties for the *mainContent* div. Notice how the View Set Properties Only checkbox (upper right) makes it easy to see what styles are set for the current rule. Also notice in the main “Artboard” how the grid rows and columns are indicated, including sliders (circled) to manipulate rows and columns directly in the artboard.

Showing this and the rest of the styling—going down into each level of the markup and creating appropriate styles in the appropriate media queries for the view states—is best done in video. Video 2-1, which is available as part of this book’s downloadable companion content, shows this whole process starting with the creation of the project, styling the different view states, and switching to Visual Studio (right-click the project name in Blend and select Edit In Visual Studio) to run the app in the simulator as a verification. It also demonstrates the approximate amount of time it takes to style an app like this once you’re familiar with the tools.

The result of all this in the simulator looks just like the wireframes—see Figures 2-12 through 2-14—and all the styling is entirely contained within the appropriate media queries of *default.css*. Most importantly, the way Blend shows us the results in real time is an enormous time-saver over fiddling with the CSS and running the app all over again, a painful process that I’m sure you’re familiar with! (And the time savings are even greater with Interactive Mode; see Video 4-1 in the companion content.)

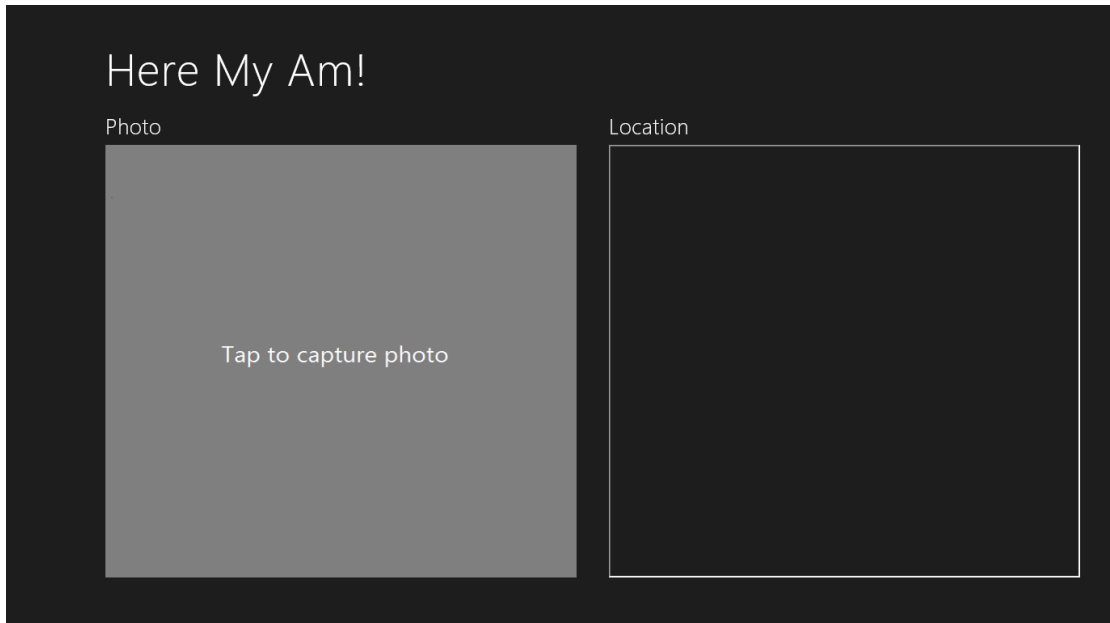


FIGURE 2-12 Full-screen landscape view.

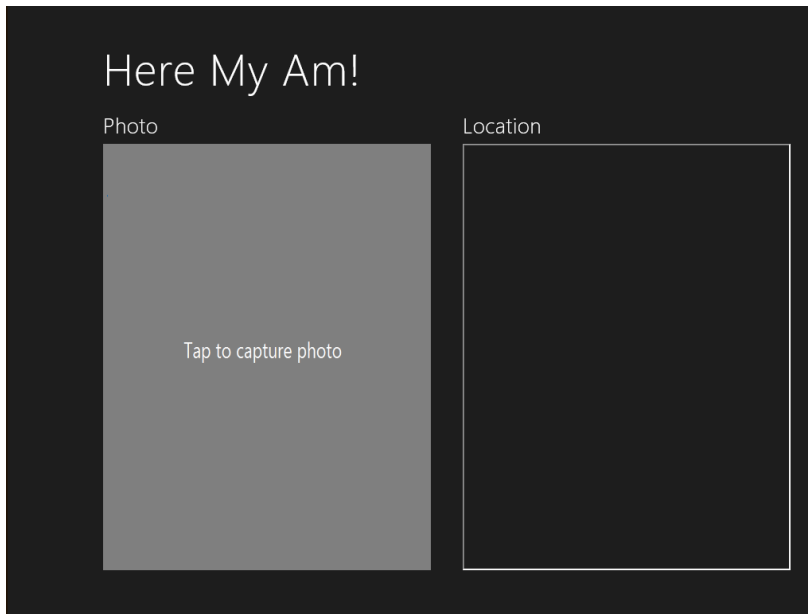


FIGURE 2-13 Filled view (landscape only).

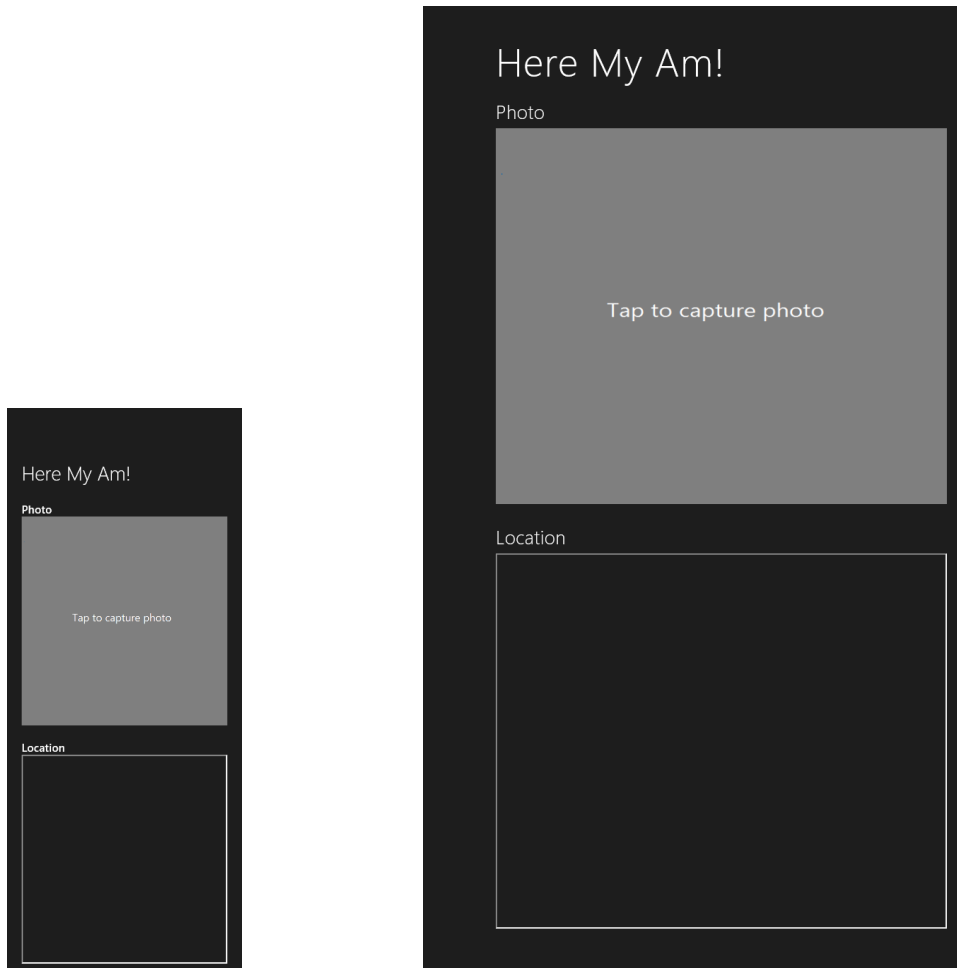


FIGURE 2-14 Snapped view (landscape only) and full-screen portrait view.

Adding the Code

Let's complete the implementation now in Visual Studio. Again, right-click the project name in Blend's Project tab and select Edit In Visual Studio if you haven't already. Note that if your project is already loaded into Visual Studio, when you switch to it, it will (by default) prompt you to reload changed files. Say yes.¹² At this point, we have the layout and styles for all the necessary view states, and our code doesn't need to care about any of it except to make some minor refinements, as we'll see in a moment.

What this means is that, for the most part, we can just write our app's code against the markup and

¹² On the flip side, note that Blend doesn't automatically save files going in and out of Interactive Mode. If you make a change to the same file open in Visual Studio, switch to Blend, and reload the file, you can lose changes.

not against the markup plus styling, which is, of course, a best practice with HTML/CSS in general. Here are the features that we'll now implement:

- A Bing maps control in the Location section showing the user's current location. In this case we'll want to adjust the zoom level of the map in snapped view to account for the smaller display area. We'll just show this map automatically, so there's no control to start this process.
- Use the WinRT APIs for camera capture to get a photograph in response to a tap on the Photo `img` element.
- Provide the photograph and the location data to the Share charm when the user invokes it.

Figure 2-15 shows what the app will look like when we're done.

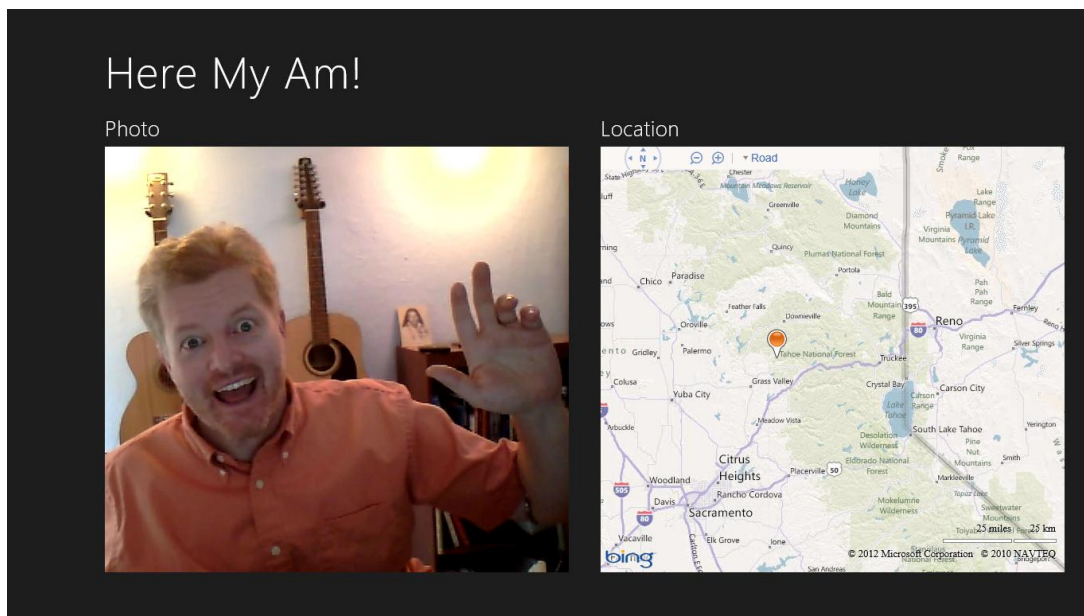


FIGURE 2-15 The Here My Am! app in its completed state (though I zoomed out the map so you can't quite tell exactly where I live!).

Creating a Map with the Current Location

For the map, we're using a Bing maps web control instantiated through the `map.html` page that's loaded into an `iframe` of the main page. This page loads the Bing Maps control script from a remote source and thus runs in the web context. Note that we could also employ the [Bing Maps for WinRT apps extension](#) that provides script we can load into the local context. For the time being, I want to use the remote script approach because it gives us an opportunity to work with web content and the web context in general, something that I'm sure you'll want to understand for your own apps. We'll switch

to the local control in Chapter 8, “State, Settings, Files, and Documents.”

That said, let’s put `map.html` in an `html` folder. Right-click the project and select Add/New Folder (entering **html** to name it). Then right-click that folder, select Add/New Item..., and then select HTML Page. Once the new page appears, replace its contents with the following:¹³

```
<!DOCTYPE html>
<html>
  <head>
    <title>Map</title>
    <script type="text/javascript"
      src="http://ecn.dev.virtualearth.net/mapcontrol/mapcontrol.ashx?v=7.0"></script>

    <script type="text/javascript">
      //Global variables here
      var map = null;

      document.addEventListener("DOMContentLoaded", init);
      window.addEventListener("message", processMessage);

      //Generic function to turn a string in the syntax { functionName: ..., args: [...] }
      //into a call to the named function with those arguments. This constitutes a generic
      //dispatcher that allows code in an iframe to be called through postMessage.
      function processMessage(message) {
        //Verify data and origin (in this case the local context page)
        if (!message.data || message.origin !== "ms-appx://" + document.location.host) {
          return;
        }

        var call = JSON.parse(message.data);

        if (!call.functionName) {
          throw "Message does not contain a valid function name.";
        }

        var target = this[call.functionName];

        if (typeof target !== 'function') {
          throw "The function name does not resolve to an actual function";
        }

        return target.apply(this, call.args);
      }

      function notifyParent(event, args) {
        //Add event name to the arguments object and stringify as the message
        args["event"] = event;
        window.parent.postMessage(JSON.stringify(args),
```

¹³ Note that you should replace the `credentials` inside the `init` function with your own key obtained from <https://www.bingmapsportal.com/>.

```

        "ms-appx://" + document.location.host);
    }

    //Create the map (though the namespace won't be defined without connectivity)
    function init() {
        if (typeof Microsoft == "undefined") {
            return;
        }

        map = new Microsoft.Maps.Map(document.getElementById("mapDiv"), {
            //NOTE: replace these credentials with your own obtained at
            //http://msdn.microsoft.com/en-us/library/ff428642.aspx
            credentials: "AhTTN0ioICXvPRPUdr0_NAYWj64MuGK2msfRendz_fL9B1U6LGDymy20hbGj7vhA",
            //zoom: 12,
            mapTypeId: Microsoft.Maps.MapTypeId.road
        });

    }

    function pinLocation(lat, long) {
        if (map === null) {
            throw "No map has been created";
        }

        var location = new Microsoft.Maps.Location(lat, long);
        var pushpin = new Microsoft.Maps.Pushpin(location, { draggable: true });

        Microsoft.Maps.Events.addHandler(pushpin, "dragend", function (e) {
            var location = e.entity.getLocation();
            notifyParent("locationChanged",
                { latitude: location.latitude, longitude: location.longitude });
        });

        map.entities.push(pushpin);
        map.setView({ center: location, zoom: 12, });
        return;
    }

    function setZoom(zoom) {
        if (map === null) {
            throw "No map has been created";
        }

        map.setView({ zoom: zoom });
    }
}
</script>
</head>
<body>
    <div id="mapDiv"></div>
</body>
</html>

```

Note that the JavaScript code here could be moved into a separate file and referenced with a relative path, no problem. I've chosen to leave it all together for simplicity.

At the top of the page you'll see a remote script reference to the Bing Maps control. We can reference remote script here because the page is loaded in the web context within the `iframe` (`ms-appx-web://` in default.html). You can then see that the `init` function is called on `DOMContentLoaded` and creates the map control. Then we have a couple of other methods, `pinLocation` and `setZoom`, which can be called from the main app as needed.

Of course, because this page is loaded in an `iframe` in the web context, we cannot simply call those functions directly from our app code. We instead use the HTML5 `postMessage` function, which raises a message event within the `iframe`. This is an important point: the local and web contexts are kept separate so that arbitrary web content cannot drive an app or access WinRT APIs. The two contexts enforce a boundary between an app and the web that can only be crossed with `postMessage`.

In the code above, you can see that we pick up such messages and pass them to the `processMessage` function, a little generic function that turns a JSON string into a local function call, complete with arguments.

To see how this works, let's look at how we call `pinLocation` from within default.js. To make this call, we need some coordinates, which we can get from the WinRT Geolocation APIs. We'll do this within the `onactivated` handler, so the user's location is just set on startup (and saved in the `lastPosition` variable sharing later on):

```
//Drop this after the line: WinJS.strictProcessing();
var lastPosition = null;

//Place this after args.setPromise(WinJS.UI.processAll());
var gl = new Windows.Devices.Geolocation.Geolocator();

gl.getGeopositionAsync().done(function (position) {
    //Save for share
    lastPosition = { latitude: position.coordinate.latitude,
                    longitude: position.coordinate.longitude };

    callFrameScript(document.frames["map"], "pinLocation",
                    [position.coordinate.latitude, position.coordinate.longitude]);
});
```

where `callFrameScript` is just a little helper function to turn the target element, function name, and arguments into an appropriate `postMessage` call:

```
//Place this before app.start();
function callFrameScript(frame, targetFunction, args) {
    var message = { functionName: targetFunction, args: args };
    frame.postMessage(JSON.stringify(message), "ms-appx-web://" + document.location.host);
}
```

A few key points about this code. First, to obtain coordinates, you can use the WinRT geolocation API or the HTML5 geolocation API. The two are almost equivalent, with slight differences described in Appendix B, "Comparing Overlapping WinRT and HTML5 APIs." The API exists in WinRT because other

supported languages (like C# and C++) don't have access to the HTML5 geolocation APIs, and because we're primarily focused on the WinRT APIs in this book, we'll just use functions in the `Windows.Devices.Geolocation` namespace.

Next, in the second parameter to `postMessage` you see a combination of `ms-appx[-web]://` with `document.location.host`. This essentially means "the current app from the local [or web] context," which is the appropriate origin of the message. Notice that we use the same value to check the origin when receiving a message: the code in `map.html` verifies it's coming from the app's local context, whereas the code in `default.js` verifies that it's coming from the app's web context. Always make sure to check the origin appropriately.

Finally, the call to `getGeopositionAsync` has an interesting construct, wherein we make the call and chain this function called `done` onto it, whose argument is another function. This is a very common pattern we'll see while working with WinRT APIs, as any API that might take longer than 50ms to complete runs asynchronously. This conscious decision was made so that the API surface area led to fast and fluid apps by default.

In JavaScript, such APIs return what's called a *promise* object, which represents results to be delivered at some time in the future. Every promise object has a `done` method whose first argument is the function to be called upon completion. It can also take two optional functions to wire up progress and error handlers as well. We'll see more about promises as we progress through this book, such as the `then` function that's just like `done` but allows further chaining (which we'll see in Chapter 3).

The argument passed to the *completed handler* (a function pass as the first argument to `done`) contains the results of the async call, which in our example above is a `Windows.Geolocation.Geoposition` object containing the last reading. (When reading the docs for an async function, you'll see that the return type is listed like `IAsyncOperation<Geoposition>`. The name within the `<>` indicates the actual data type of the results, so you'll normally follow the link to that topic for the details.) The coordinates from this reading are what we then pass to the `pinLocation` function within the `iframe`, which in turn creates a pushpin on the map at those coordinates and then centers the map view at that same location.¹⁴

One final note about async APIs. Within the WinRT API, all async functions have "Async" in their names. Because this isn't common practice within *JavaScript* toolkits or the DOM API, async functions within WinJS don't use that suffix. In other words, WinRT is designed to be language-neutral, but WinJS is designed to follow typical JavaScript conventions.

Oh Wait, the Manifest!

Now you may have tried the code above and found that you get an "Access is denied" exception when you try to call `getGeopositionAsync`. Why is this? Well, the exception tells us: we neglected to set the

¹⁴ The pushpin itself is draggable, but to no effect at present. See the section "Extra Credit: Receiving Messages from the iframe" later in this chapter for how we can pick up location changes from the map.

Geolocation capability in the manifest. Without that capability set, calls like this that depend on that capability will throw an exception.

We were running in the debugger, so that exception was kindly shown to us. If you run the app outside of the debugger—try it from the tile that should be on your Start page—you’ll see that it just terminates without showing anything but the splash screen. This is the default behavior for an unhandled exception. To prevent that behavior, add an error-handling function as the second parameter to the async promise’s `done` method:

```
gl.getGeopositionAsync().then(function (position) {
    //...
}, function(error) {
    console.log("Unable to get location.");
});
```

The `console.log` function writes a string to the *JavaScript Console window* in Visual Studio, which is obviously a good idea. Now run the app outside the debugger and you’ll see that it comes up, because the exception is now considered “handled.” In the debugger, set a breakpoint on the `console.log` line inside and you’ll hit that breakpoint after the exception appears (and you press Continue).

If the exception dialog gets annoying, you can control which exceptions pop up like this in the Debug > Exceptions dialog box (shown in Figure 2-16) within JavaScript Runtime Exceptions. If you uncheck User-unhandled, you won’t get a dialog when the exception occurs. (That dialog also has a checkbox for this as well.)

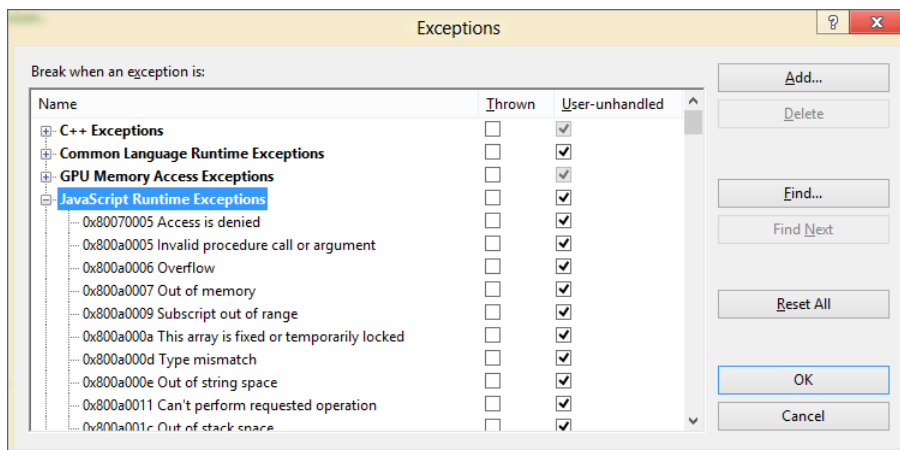


FIGURE 2-16 JavaScript run-time exceptions in the Debug/Exceptions dialog of Visual Studio.

Back to the capability: to get the proper behavior for this app, open `package.appxmanifest` in your project, select the Capabilities tab, and check Location, as shown in Figure 2-17.

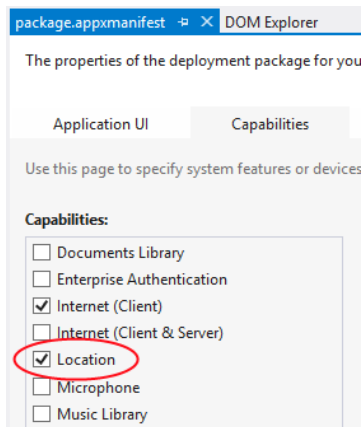


FIGURE 2-17 Setting the Location capability in Visual Studio’s manifest editor. (Note that Blend supports editing the manifest only as XML.)

Now, even when we declare the capability, geolocation is still subject to user consent, as mentioned in Chapter 1. When you first run the app with the capability set, then, you should see a popup like Figure 2-18. If the user blocks access here, the error handler will again be invoked as the API will throw an Access denied exception.



FIGURE 2-18 A typical consent popup, reflecting the user’s color scheme, that appears when an app first tries to call a brokered API (geolocation in this case). If the user blocks access, the API will fail, but the user can later change consent in the Settings/Permissions panel.

Sidebar: How Do I Reset User Consent for Testing?

While debugging, you might notice that this popup appears only once, even across subsequent debugging sessions. To clear this state, invoke the Settings charm in the running app and select Permissions, and you’ll see toggle switches for all the relevant capabilities. If for some reason you can’t run the app at all, go to the Start screen and uninstall the app from its tile. You’ll then see the popup when you next run the app.

Note that there isn’t a notification when the user changes these Permission settings. The app can detect a change only by attempting to use the API again. We’ll revisit this subject in Chapter 8.

Capturing a Photo from the Camera

In a slightly twisted way, I hope the idea of adding camera capture within a so-called “quickstart”

chapter has raised serious doubts in your mind about this author's sanity. Isn't that going to take a whole lot of code? Well, it *used* to, but it doesn't on Windows 8. All the complexities of camera capture have been nicely encapsulated within the [Windows.Media.Capture](#) API to such an extent that we can add this feature with only a few lines of code. It's a good example of how a little dynamic code like JavaScript combined with well-designed WinRT components—both those in the system and those you can write yourself—make a very powerful combination!

To implement this feature, we first need to remember that like geolocation, the camera is a privacy-sensitive device and must also be declared in the manifest, as shown in Figure 2-19.

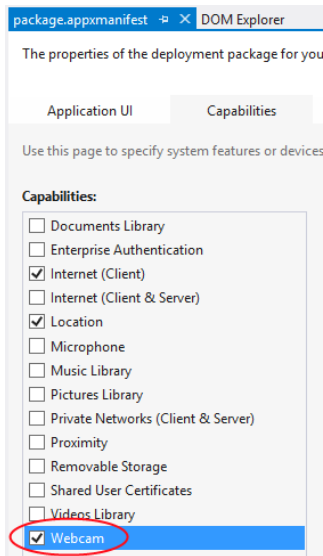


FIGURE 2-19 The camera capability in Visual Studio's manifest editor.

On first use of the camera at run time, you'll see a consent dialog, as with geolocation, like the one shown in Figure 2-20.



FIGURE 2-20 Popup for obtaining the user's consent to use the camera. You can control these through the Settings/Permissions panel at any time.

Next we need to wire up the `img` element to pick up a tap gesture. For this we simply need to add an event listener for `click`, which works for all forms of input (touch, mouse, and stylus), as we'll see in Chapter 9, "Input and Sensors":

```
var image = document.getElementById("photo");
image.addEventListener("click", capturePhoto.bind(image));
```

Here we're providing `capturePhoto` as the event handler, and using the function object's `bind` method to make sure the `this` object inside `capturePhoto` is bound directly to the `img` element. The result is that the event handler can be used for any number of elements because it doesn't make any references to the DOM itself:

```
//Place this under var lastPosition = null;
var lastCapture = null;

//Place this after callFrameScript
function capturePhoto() {
    //Due to the .bind() call in addEventListener, "this" will be the image element,
    //but we need a copy for the async completed handler below.
    var that = this;

    var captureUI = new Windows.Media.Capture.CameraCaptureUI();

    //Indicate that we want to capture a PNG that's no bigger than our target element --
    //the UI will automatically show a crop box of this size
    captureUI.photoSettings.format = Windows.Media.Capture.CameraCaptureUIPhotoFormat.png;
    captureUI.photoSettings.croppedSizeInPixels =
        { width: this.clientWidth, height: this.clientHeight };

    captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
        .done(function (capturedFile) {
            //Be sure to check validity of the item returned; could be null if the user canceled.
            if (capturedFile) {
                lastCapture = capturedFile; //Save for Share
                that.src = URL.createObjectURL(capturedFile, {oneTimeOnly: true});
            }
        }, function (error) {
            console.log("Unable to invoke capture UI.");
        });
}
```

We *do* need to make a local copy of `this` within the `click` handler, though, because once we get inside the async completed function (see the function inside `captureFileAsync.done`) we're in a new function scope and the `this` object will have changed. The convention for such a copy of `this` is to call it `that`. Got that?

To invoke the camera UI, we only need create an instance of `Windows.Media.Capture.CameraCaptureUI` with `new` (a typical step to instantiate dynamic WinRT objects), configure it with the desired format and size (among may other possibilities as discussed in Chapter 10, "Media"), and then call `captureFileAsync`. This will check the manifest capability and prompt the user for consent, if necessary.

This is an async call, so we hook a `.done` on the end with a completed handler, which in this case will receive a `Windows.Storage.StorageFile` object. Through this object you can get to all the raw image data you want, but for our purpose we simply want to display it in the `img` element.

Fortunately, that's super-easy as well! You can hand a `StorageFile` object to the

`URL.createObjectURL` method and get back an URI that can be directly assigned to the `img.src` attribute. Voila! The captured photo appears!¹⁵

Note that `captureFileAsync` will call the completed handler if the UI was successfully invoked but the user hit the back button and didn't actually capture anything. This is why the extra check is there for the validity of `capturedFile`. An error handler on the promise will, for its part, pick up failures to invoke the UI in the first place, but note that a denial of consent will show a message in the capture UI directly (see Figure 2-21), so it's unnecessary to have an error handler for that purpose with this particular API. In most cases, however, you'll want to have an error handler in place for `async` calls.

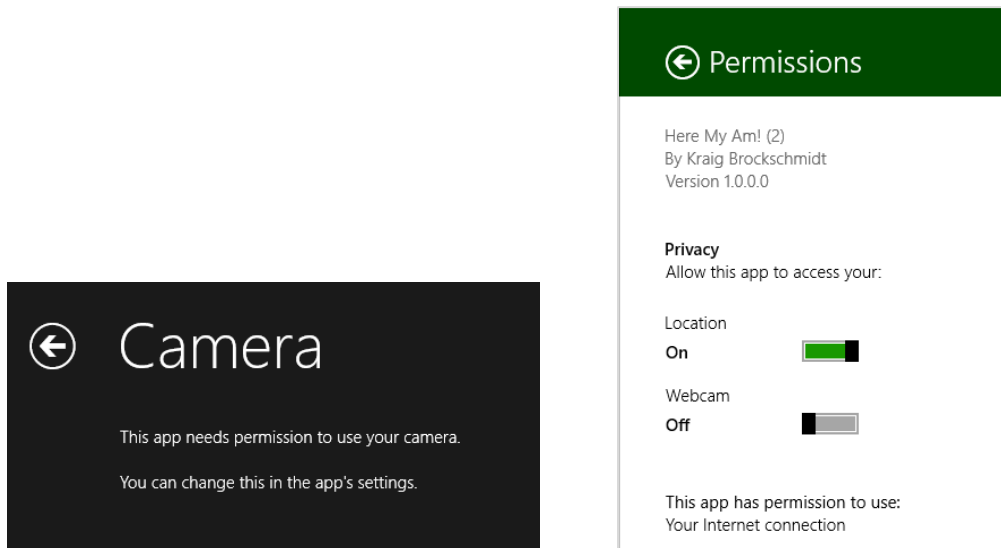


FIGURE 2-21 The camera capture UI's message when consent is denied (left); you can change permissions through the Settings Charm > Permissions pane (right).

Sharing the Fun!

Taking a goofy picture of oneself is fun, of course, but sharing the joy with the rest of the world is even better. Up to this point, however, sharing information through different social media apps has meant using the specific APIs of each service. Workable, but not scalable.

Windows 8 has instead introduced the notion of the *share contract*, which is used to implement the Share charm with as many apps as participate in the contract. Whenever you're in an app and invoke Share, Windows sends whatever data that *source* app makes available (if any) to whatever other *target* app the user selects (from a list of those whose manifests identify them as targets). The contract is an

¹⁵ The `{oneTimeOnly: true}` parameter indicates that the URI is not reusable and should be revoked via `URL.revokeObjectURL` when it's no longer used, as when we replace the `img src` with a new picture. Without this, we would leak memory with each new picture. If you've used `URL.createObjectURL` in the past, you'll see that the second parameter is now a property bag, which aligns with the most recent W3C spec.

abstraction that sits between the two, so the source and target apps never need to know anything about each other.

This makes the whole experience all the richer as the user installs more share-capable apps, and it doesn't limit sharing to only well-known social media scenarios. What's also beautiful in the overall experience is that the user never leaves the original app to do sharing—the share target app shows up in its own view as an overlay that only partially obscures the source app. This way, the user immediately returns to that source app when the sharing is completed, rather than having to switch back to that app manually.

So instead of adding code to our app to share the photo and location to a particular target, like Facebook, we only need to package the data appropriately when Windows asks for it.

That asking comes through the `daterequested` event sent to the `Windows.ApplicationModel.DataTransfer.DataTransferManager` object. First we just need to set up an appropriate listener—place this code in the `onactivated` event in `default.js` after setting up the `click` listener on the `img` element:

```
var dataTransferManager =  
    Windows.ApplicationModel.DataTransfer.DataTransferManager.getForCurrentView();  
dataTransferManager.addEventListener("daterequested", provideData);
```

The idea of a *current view* is something that we'll see pop up now and then. It reflects that an app can be launched for different reasons—such as servicing a contract—and thus presents different underlying pages or views to the user at those times. These views (unrelated to the snap/fill/etc. view *states*) can be active simultaneously. To thus make sure that your code is sensitive to these scenarios, certain APIs return objects appropriate for the current view of the app as we see here.

For this event, the handler receives a `Windows.ApplicationModel.DataTransfer.DataRequest` object in the event args (`e.request`), which in turn holds a `DataPackage` object (`e.request.data`). To make data available for sharing, you populate this data package with the various formats you have available. (We've saved these in `lastPosition` and `lastCapture`.) In our case, we make sure we have position and a photo, then fill in text and image properties:

```
//Drop this in after capturePhoto  
function provideData(e) {  
    var request = e.request;  
    var data = request.data;  
  
    if (!lastPosition || !lastCapture) {  
        //Nothing to share, so exit  
        return;  
    }  
  
    data.properties.title = "Here My Am!";  
    data.properties.description = "At (" +  
        lastPosition.latitude + ", " + lastPosition.longitude + ")";  
  
    //When sharing an image, include a thumbnail
```

```

var streamReference =
    Windows.Storage.Streams.RandomAccessStreamReference.createFromFile(lastCapture);
data.properties.thumbnail = streamReference;

//It's recommended to always use both setBitmap and setStorageItems for sharing a single image
//since the target app may only support one or the other.

//Put the image file in an array and pass it to setStorageItems
data.setStorageItems([lastCapture]);

//The setBitmap method requires a RandomAccessStream.
data.setBitmap(streamReference);
}

```

The latter part of this code is pretty standard stuff for sharing a file-based image (which we have in `lastCapture`). I got most of this code, in fact, directly from the SDK's [Share content source app sample](#), which we'll look at more closely in Chapter 12, "Contracts."

With this last addition of code, and a suitable sharing target installed (such as the SDK's [Share content target app sample](#), as shown in Figure 2-22), we now have a very functional app—in all of 35 lines of HTML, 125 lines of CSS, and less than 100 lines of JavaScript!

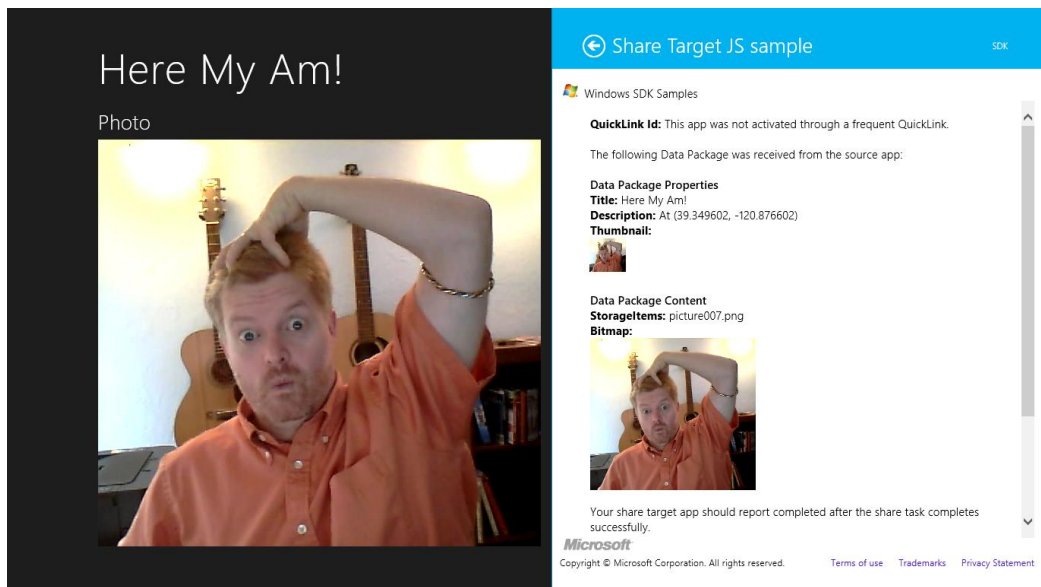


FIGURE 2-22 Sharing (monkey-see, monkey-do!) to the Share target sample in the Windows SDK. Share targets appear as a partial overlay on top of the current app, so the user never leaves the app context.

Extra Credit: Receiving Messages from the iframe

There's one more piece I've put into Here My Am! to complete the basic interaction between app and `iframe` content: the ability to post messages from the `iframe` back to the main app. In our case, we

want to know when the location of the pushpin has changed so that we can update `lastPosition`.

First, here's a simple utility function I added to `map.html` to encapsulate the appropriate `postMessage` calls to the app from the `iframe`:

```
function function notifyParent(event, args) {  
    //Add event name to the arguments object and stringify as the message  
    args["event"] = event;  
    window.parent.postMessage(JSON.stringify(args), "ms-appx://" + document.location.host);  
}
```

This function basically takes an event name, adds it to whatever object is given containing parameters, and then stringifies the whole bit and posts it back to the parent.

When a pushpin is dragged, Bing maps raises a `dragend` event, which we'll wire up and handle in the `setLocation` function just after the pushpin is created (also in `map.html`):

```
var pushpin = new Microsoft.Maps.Pushpin(location, { draggable: true });  
  
Microsoft.Maps.Events.addHandler(pushpin, "dragend", function (e) {  
    var location = e.entity.getLocation();  
    notifyParent("locationChanged",  
        { latitude: location.latitude, longitude: location.longitude });  
});
```

Back in `default.js` (the app), we add a listener for incoming messages inside `app.onactivated`:
`window.addEventListener("message", processFrameEvent);`

where the `processFrameEvent` handler looks at the event in the message and acts accordingly:

```
function processFrameEvent (message) {  
    //Verify data and origin (in this case the web context page)  
    if (!message.data || message.origin !== "ms-appx-web://" + document.location.host) {  
        return;  
    }  
  
    if (!message.data) {  
        return;  
    }  
  
    var eventObj = JSON.parse(message.data);  
  
    switch (eventObj.event) {  
        case "locationChanged":  
            lastPosition = { latitude: eventObj.latitude, longitude: eventObj.longitude };  
            break;  
  
        default:  
            break;  
    }  
};
```

Clearly, this is more code than we'd need to handle a single message or event from an `iframe`, but I

wanted to give you something that could be applied more generically in your own apps.

The Other Templates

In this chapter we've worked only with the Blank App template so that we could understand the basics of writing a WinRT app without any other distractions. In Chapter 3, we'll look more deeply at the anatomy of apps through a few of the other templates, yet we won't cover them all. We'll close this chapter, then, with a short introduction to these very handy tools.

Fixed Layout Template

"A project for a Windows Store app that scales using a fixed aspect ratio layout." (Blend/Visual Studio description)

What we've seen so far are examples of apps that adapt themselves to changes in display area by adjusting the layout. In Here My Am!, for instance, we used CSS grids with self-adjusting areas (those 1fr's in rows and columns). This works great for apps with content that is suitably resizable as well as apps that can show additional content when there's more room, such as more news headlines or items from a search.

Other kinds of apps are not so flexible, such as games where the aspect ratio of the playing area needs to stay constant. (It would not be fair if players on larger screens got to see more of the game!) So, when the display area changes—either from view states or a change in display resolution—they do better to scale themselves up or down rather than adjust their layout.

The Fixed Layout template provides the basic structure for such an app, just like the Blank template provides for a flexible app. The key piece is the `WinJS.UI.ViewBox` control, which automatically takes care of scaling its contents while maintaining the aspect ratio:

```
<body>
  <div data-win-control="WinJS.UI.ViewBox">
    <div class="fixedlayout">
      <p>Content goes here</p>
    </div>
  </div>
</body>
```

In `default.css`, you can see that the `body` element is styled as a CSS flexbox centered on the screen and the `fixedLayout` element is set to 1024x768 (the minimum size for the fullscreen-landscape and filled view states). Within the child `div` of the `ViewBox`, then, you can safely assume that you'll always be working with these fixed dimensions. The `ViewBox` will scale everything up and provide letterboxing as necessary.

Note that such apps might not be able to support an interactive snapped state; a game, for example, will not be playable when scaled down. In this case an app can simply pause the game and try

to unsnap itself when the user taps it again. We'll revisit this in Chapter 6.

Navigation Template

"A project for a Windows Store app that has predefined controls for navigation." (Blend/Visual Studio description)

The Navigation template builds on the Blank template by adding support for page navigation. As discussed in Chapter 1, WinRT apps written in HTML/JavaScript are best implemented by having a single HTML page container into which other pages are dynamically loaded. This allows for smooth transitions (as well as animations) between those pages and preserves the script context.

This template, and the others that remain, employ a Page Navigator control that facilitates loading (and unloading) pages in this way. You need only create a relatively simple structure to describe each page and its behavior. We'll see this in Chapter 3.

In this model, `default.html` is little more than a simple container, with everything else in the app coming through subsidiary pages. The Navigation template creates only one subsidiary page, yet it establishes the framework for how to work with multiple pages.

Grid Template

"A three-page project for a Windows Store app that navigates among grouped items arranged in a grid. Dedicated pages display group and item details." (Blend/Visual Studio description)

Building on the Navigation template, the Grid template provides the basis for apps that will navigate collections of data across multiple pages. The home page shows grouped items within the collection, from which you can then navigate into the details of an item or into the details of a group and its items (from which you can then go into item details as well).

In addition to the navigation, the Grid template also shows how to manage collections of data through the `WinJS.Binding.List` class, a topic we'll explore much further in Chapter 5, "Collections and Collection Controls." It also provides the structure for an app bar and shows how to simplify the app's behavior in snap view.

The name of the template, by the way, derives from the particular "grid" *layout* used to display the collection, not from the CSS grid.

Split Template

"A two-page project for a Windows Store app that navigates among grouped items. The first page allows group selection while the second displays an item list alongside details for the selected item." (Blend/Visual Studio description)

This last template also builds on the Navigation template and works over a collection of data. Its home page displays a list of groups, rather than grouped items as with the Grid template. Tapping a

group then navigates to a group detail page that is split into two sides (hence the template name). The left side contains a vertically panning list of items; the right side shows details for the currently selected item.

Like the Grid template, the Split template provides an app bar structure and handles both snap and portrait views intelligently. That is, because vertically oriented views don't lend well to splitting the display (contrary to the description above!), the template shows how to switch to a page navigation model within those view states to accomplish the same ends.

What We've Just Learned

- How to create a new WinRT app from the Blank app template.
- How to run an app inside the local debugger and within the simulator.
- The features of the simulator.
- The basic project structure for WinRT apps, including WinJS references.
- The core activation structure for an app.
- The role and utility of design wireframes in app development, including the importance of designing for all view states.
- How to quickly and efficiently add styling to an app's markup in Blend for Visual Studio.
- How to safely use web content (such as Bing maps) within an `iframe` and communicate between that page and the app.
- How to use the WinRT APIs, especially async methods involving promises but also geolocation and camera capture.
- The importance of manifest capabilities in being able to use certain WinRT APIs.
- How to share data through the Share contract.
- The kinds of apps supported through the other app templates: Fixed Layout, Navigation, Grid, and Split.

Chapter 3

App Anatomy and Page Navigation

During the early stages of writing this book, I was also working closely with a contractor to build a house for my family. While I wasn't on site every day managing the whole effort, I was certainly involved in most decision-making throughout the home's many phases, and I occasionally participated in the construction itself.

In the Sierra Nevada foothills of California, where I live, the frame of a house is built with the plentiful local wood, and all the plumbing and wiring has to be in the walls before installing insulation and wallboard (aka sheetrock). It amazed me how long it took to complete that infrastructure. The builders spent a lot of time adding little blocks of wood here and there to make it much easier for them to do the finish work later on (like hanging cabinets), and lots of time getting the wiring and plumbing put together properly. All of this becomes completely invisible to the eye once the wallboard is on and the finish work is in place.

But then, imagine what the house would be like without such careful attention to structural details. Imagine having some light switches that just didn't work or controlled the wrong fixtures. Imagine if the plumbing leaked. Imagine if cabinets and trim started falling off the walls after a week or two of living in the house. Even if the house managed to pass final inspection, such flaws would make it almost unlivable, no matter how beautiful it might appear at first sight. It would be like a few of the designs of the famous architect Frank Lloyd Wright: very interesting architecturally and aesthetically pleasing, yet thoroughly uncomfortable to actually live in.

Apps are very much the same story—I've marveled, in fact, just how many similarities exist between the two endeavors! That is, an app might be visually beautiful, even stunning, but once you really start using it day to day, a lack of attention on the fundamentals will become painfully apparent.

This chapter, then, is about those fundamentals: the core foundational structure of an app upon which you can build something that can look beautiful *and* really work well. We'll first complete our understanding of the hosted environment and then look at activation (how apps get running) and lifecycle transitions. We'll then look at page navigation within an app, and we'll see a few other important considerations along the way, such as working with multiple async operations.

Let me offer you advance warning that this is an admittedly longer and more intricate chapter than many that follow, since it specifically deals with the software equivalents of framing, plumbing, and wiring. With our house, I can completely attest that installing the lovely light fixtures my wife picked out seemed, in the moment, much more satisfying than the framing I'd done months earlier. But now, actually *living* in the house, I have a deep appreciation for all the nonglamorous work that went into it. It's a place I want to be, a place in which my family and I are delighted, in fact, to spend the majority of our lives. And is that not how you want your customers to feel about your apps? Absolutely! Knowing the delight that a well-architected app can bring to your customers, let's dive in and find our own

delight in exploring the intricacies!

Local and Web Contexts within the App Host

As described in Chapter 1, “The Life Story of a WinRT app,” apps written with HTML, CSS, and JavaScript are not directly executable apps like their compiled counterparts written in C#, Visual Basic, or C++. In our app packages, there are no .EXEs, just .html, .css, and .js files (plus resources, of course) that are, plain and simple, nothing but text. So something has to turn all this text that defines an app into something that’s actually running in memory. That something is again the *app host*, *wwahost.exe*, which creates what we call the *hosted environment* for WinRT apps.

We’ve already covered most of the characteristics of the hosted environment in Chapter 1 and Chapter 2, “Quickstart”:

- The app host (and the apps in it) use brokered access to sensitive resources.
- Though the app host provides an environment very similar to that of Internet Explorer 10, there are a number of changes to the DOM API, documented on [HTML and DOM API changes list](#) and [HTML, CSS, and JavaScript features and differences](#).
- HTML content in the app package can be loaded into the *local* or *web context*, depending on the `ms-appx:///` and `ms-appx-web:///` scheme used to reference that content (the third / again means “in the app package”). Remote content (referred to with `http[s]://`) always runs in the web context.
- The local context has access to the WinRT API, among other things, whereas the web context is allowed to load and execute remote script but cannot access WinRT.
- ActiveX plug-ins are generally not allowed in either context.
- The HTML5 `postMessage` function can be used to communicate between an `iframe` and its containing parent across contexts. This can be useful to execute remote script within the web context and pass the results to the local context; script acquired in the web context should not be itself passed to the local context and executed there. (Windows Store policy actually disallows this, and apps submitted to the Store will be analyzed for such practices.)
- Further specifics can be found on [Features and restrictions by context](#), including which parts of WinJS don’t rely on WinRT and can thus be used in the web context. (WinJS, by the way, cannot be used on web *pages* outside of an app.)

Now what we’re really after in this chapter is not so much these characteristics themselves but their impact on the structure of an app. First and foremost is that an app’s home page (the one you point to

in the manifest in the Start page field of the Application UI tab¹⁶) *always* runs in the local context, and any page to which you navigate directly (`<a href>` or `document.location`) must also be in the local context. (You can try otherwise, but the app host will display an interesting “not supported” message right inside your app!) Those pages, however, can contain `iframe` elements in either context, depending on which scheme you use.

A local context page can contain an `iframe` in either local or web context, provided that the `src` attribute refers to content in the app package (and by the way, programmatic read-only access to your package contents is obtained via `Windows.ApplicationMode.Package.Current.InstalledLocation`). Referring to any other location (`http[s]://` or other protocols) will always place the `iframe` in the web context.

```
<!-- iframe in local context with source in the app package -->
<!-- this form is only allowed from inside the local context -->
<iframe src=" /frame-local.html"></iframe>
<iframe src="ms-appx:///frame-local.html"></iframe>

<!-- iframe in web context with source in the app package -->
<iframe src="ms-appx-web:///frame-web.html"></iframe>

<!-- iframe with an external source automatically assigns web context -->
<iframe src="http://www.bing.com"></iframe>
```

Also, if you use an `` tag with `target` pointing to an `iframe`, the scheme in `href` determines the context.

A web context page, for its part, can contain an `iframe` only in the web context; for example, the last two `iframe` elements above are allowed, whereas the first two are not. You can also use `ms-appx-web:///` within the web context to refer to other content within the app package, such as images.

Although not commonly done within WinRT apps for reasons we’ll see later in this chapter, similar rules apply with page-to-page navigation using `<a href>` or `document.location`. Since the whole scene here can begin to resemble overcooked spaghetti, the exact behavior for these variations and for `iframes` is described in the following table:

Target	Result in Local Context Page	Result in Web Context Page
<code><iframe src="ms-appx:///"></code>	<code>iframe</code> in local context	Not allowed
<code><iframe src="ms-appx-web:///"></code>	<code>iframe</code> in web context	<code>iframe</code> in web context
<code><iframe src="http[s]:// "></code> or other scheme	<code>iframe</code> in web context	<code>iframe</code> in web context
<code></code> <code><iframe name="myFrame"></code>	<code>iframe</code> in local or web context depending in [uri]	<code>iframe</code> in web context; [uri] cannot begin with <code>ms-appx</code> .
<code></code>	Navigates to page in local context	Not allowed unless explicitly specified (see below)
<code></code>	Not allowed	Navigates to page in web context

¹⁶ The manifest names this the “Start page,” but I prefer “home page” to avoid confusion with the Windows Start screen.

<code></code> with any other protocol including <code>http[s]</code>	Opens default browser with [uri]	Opens default browser with [uri]
--	----------------------------------	----------------------------------

When an `iframe` is in the web context, note that its page can contains `ms-appx-web` references to in-package resources, even if the page is loaded from a remote source (`http[s]`). Such pages, of course, would not work in a browser.

The last two items in the table really mean that a WinRT app cannot navigate from its top-level page (in the local context) directly to a web context page of any kind (local or remote). That's just life in the app host! Such content must be placed in an `iframe`.

Similarly, navigating from a web context page to a local context page is not allowed by default, but you can enable this by calling the super-secret function `MSApp.addPublicLocalApplicationUri` (from code in a local page, and it actually is well-documented) for each specific URI you need:

```
//This must be called from the local context
MSApp.addPublicLocalApplicationUri("ms-appx:///frame-local.html");
```

The Direct Navigation example for this chapter gives a demonstration of this. Do be careful when the URI contains query parameters, however. For example, you don't want to allow a website to navigate to something like `ms-appx:///delete.html?file=superimportant.doc!`

One other matter that arises here is the ability to grant a web context page access to specific functions like geolocation and writing to the clipboard—things that web pages typically assume they can use. By default, the web context in a WinRT app has no access to such operating system capabilities. For example, create a new Blank project in Visual Studio with this one line of HTML in the body of `default.html`:

```
<iframe src="http://maps.bing.com" style="width:1366px; height: 768px"></iframe>
```

Then set the Location capability in the manifest (something I forgot on my first experiment with this!), and run the app. You'll see the Bing page you expect.¹⁷ However, attempting to use geolocation from within that page—clicking the locator control to the left of "World," for instance—will give you the kind of error shown in Figure 3-1.

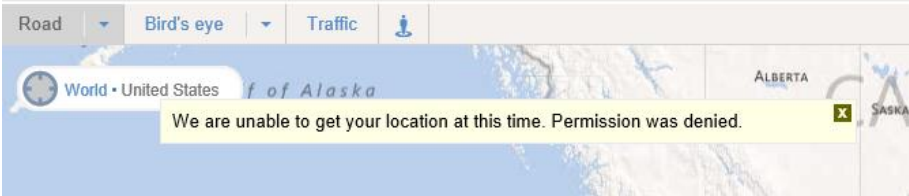


Figure 3-1 Use of brokered capabilities like geolocation from within a web context will generate an error.

Such capabilities are blocked because web content loaded into an `iframe` can easily provide the

¹⁷ If the color scheme looks odd, it's because the `iframe` is picking up styles from the default `ui-dark.css` of WinJS. Try changing that stylesheet to `ui-light.css` for something that looks more typical.

means to navigate to other arbitrary pages. From the Bing maps page used above, for example, a user can go to the Bing home page, do a search, and end up on any number of untrusted and potentially malicious pages. Whatever the case, those pages might request access to sensitive resources, and if they just generated the same user consent prompts as an app, users could be tricked into granting such access.

Fortunately, if you ask nicely, Windows will let you enable those capabilities for web pages that the app knows about. All it takes is an affidavit signed by you and sixteen witnesses, and...OK, I'm only joking! You simply need to add what are called *application content URI rules* to your manifest. Each rule says that content from some URI is known and trusted by your app and can thus act on the app's behalf. (You can also exclude URIs, which is typically done to exclude specific pages that would otherwise be included within another rule.)

Such rules are created in the Content Uri tab of Visual Studio's manifest editor, as shown in Figure 3-2. Each rule needs to be the exact URI that might be making a request, *http://www.bing.com/maps/*. Once we add that rule (as in the completed ContentUri example for this chapter), Bing maps is allowed to use geolocation. When it does so, a message dialog will appear (Figure 3-3), just as if the app had made the request. (Note: when run inside the debugger, the ContentUri example will show a Permission Denied exception on startup. *This is expected* and you can press Continue within Visual Studio; it doesn't affect the app running outside the debugger.)

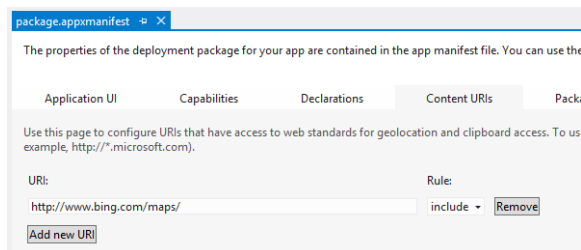


Figure 3-2 Adding a content URI to the app manifest; the contents of the text box is saved when the manifest is saved. Add New URI creates another set of controls in which to enter additional rules.

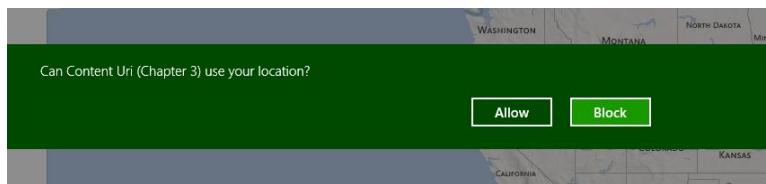


Figure 3-3 With an content URI rule in place, web content in an *iframe* acts like part of the app. This shows exactly why content URI rules are necessary to protect the user from pages unknown to the app that could otherwise trick the user into granting access to sensitive resources.

Sidebar: A Few iframe Tips and Cautions

As we're talking about *iframe* elements here, there are a couple extra tips you might find helpful

when using them. First, to prevent selection, style the `iframe` with `-ms-user-select: none` or set its `style.msUserSelect` property to `"none"` in JavaScript. Second, some web pages contain frame-breaking code that prevents the page from being loaded into an `iframe`, in which case the page will be opened in the default browser and not the app. Third, just as plug-ins aren't supported in WinRT apps, they'll also fail to load for web pages loaded into an `iframe`. In short, pulling web content that you don't own into an app is a risky business!

Furthermore, `iframe` support is not intended to let you just build an app by pulling in remote web pages. The Windows Store Certification Requirements, in fact, specifically disallow apps that are just websites—the primary app experience must take place within the app and not within web sites hosted in `iframe` elements. (See section 2.4 in those requirements.) A few key reasons for this are that websites typically aren't set up well for touch interaction (which violates requirement 3.5) and often won't work well in snapped view (violating requirement 3.6). In short, overuse of web content will likely mean that the app won't be accepted by the Store.

Referencing Content from App Data: ms-appdata

As we've seen, the `ms-appx[-web]:///` schema allow an app to navigate `iframe` elements to pages that exist inside the app package, or on the web. This begs a question: can an app point to content on the local file system that exists outside its package, such as a dynamically created file in an appdata folder? Can, perchance, an app use the `file://` protocol to navigate and/or access that content?

Well, as much as I'd love to tell you that this just works, the answer is somewhat mixed. First off, the `file://` protocol is wholly blocked by design for various security reasons, even for your appdata folders to which you otherwise have full access. (Custom protocols are also unsupported in `iframe src` URIs.) Fortunately there is a substitute, `ms-appdata://`, that fulfills part of the need. Within the local context of an app, `ms-appdata` is a shortcut to the appdata folder wherein exist local, roaming, and temp folders. So, if you created a picture called `image65.png` in your appdata local folder, you can refer to it by using `ms-appdata:///local/image65.png` (and similar forms with `roaming` and `temp`) wherever a URI can be used, including within a CSS style like `background`.

Unfortunately, the caveat—there always seems to be one with the app container!—is that `ms-appdata` can be used only for resources, namely with the `src` attribute of `img`, `video`, and `audio` elements. It cannot be used to load HTML pages, CSS stylesheets, or JavaScript, nor can it be used for navigation purposes (`iframe`, hyperlinks, etc.).

Can you do any kind of dynamic page generation, then? Well, yes: you need to load file contents and process them manually. You can get to your appdata folders through the `Windows.Storage.ApplicationData` API and go from there. To load and render a full HTML page requires that you patch up all external references and play some magic with script, but it can be done if you really want.

A similar question is whether you can generate and execute script on the fly. The answer is again qualified. Yes, you can take a JavaScript string and pass it to the `eval` or `execScript` functions. The inevitable caveat here is that automatic filtering is applied to that code that prevents injection of script

(and other risky markup) into the DOM via properties like `innerHTML` and `outerHTML`, and methods like `document.write` and `DOMParser.parseFromString`. Yet there are certainly situations where you, the developer, really know what you're doing and enjoy juggling flaming swords and running chainsaws and thus want to get around such restrictions, especially when using third-party libraries. (See the sidebar below.) Acknowledging that, Microsoft provides a mechanism to consciously circumvent all this: `MSApp.execUnsafeLocalFunction`. For all the details regarding this, refer to [Developing secure apps](#), which covers this along with a few other obscure topics (like the `sandbox` attribute for `iframes`) that I'm not including here.

And curiously enough, WinJS actually makes it *easier* for you to juggle flaming swords and running chainsaws! `WinJS.Utilities.setInnerHTMLUnsafe`, `setOuterHTMLUnsafe`, and `insertAdjacentHTMLUnsafe` are wrappers for calling DOM methods that would otherwise strip out risky content.

All that said (don't you love being aware of the details?), let's look at an example of using `ms-appdata`, which will probably be much more common in your app-building efforts.

Sidebar: Third-Party Libraries and the Hosted Environment

In general, WinRT apps can employ libraries like jQuery, Prototype, Dojo, and so forth, as noted in Chapter 1. However, there are some limitations and caveats.

First, because local context pages in an app cannot load script from remote sources, apps typically need to include such libraries in their packages unless only being used from the web context. (WinJS, mind you, doesn't need bundling because it's provided by the Windows Store—such “framework packages” are not enabled for third parties in Windows 8.)

Second, DOM API changes and app container restrictions might affect the library. For example, library functions using `window.alert` won't work. One library also cannot load another library from a remote source in the local context. Most importantly, anything in the library that assumes a higher level of trust than the app container provides, such as assuming open file system access, will have issues.

The most common issue comes up when libraries inject elements or script into the DOM (as through `innerHTML`), a widespread practice for web applications that is not generally allowed within the app container. For example, trying to create a jQuery datepicker widget (`$("#myCalendar").datepicker()`) will hurl out this kind of error. You can get around this on the app level by wrapping the code above with `MSApp.execUnsafeLocalFunction`, but that doesn't solve injections coming from deeper inside the library. In the jQuery example given here, the control can be created but clicking a date in that control generates another error.

In short, you're free to use third-party libraries so long as you're aware that they were generally written with assumptions that don't always apply within the app container. Over time, of course, fully Windows 8-compatible versions of such libraries will emerge.

Here My Am! with ms-appdata

OK! Having endured seven pages of esoterica, let's play with some real code and return to the Here My Am! app we wrote in Chapter 2. Here My Am! used the convenient `URL.createObjectURL` method to display a picture taken through the camera capture UI in an `img` element:

```
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
  .done(function (capturedFile) {
    if (capturedFile) {
      that.src = URL.createObjectURL(capturedFile);
    }
  });
```

This is all well and good, if we just take it on faith that the picture is stored somewhere—we don't really care so long as we get a URL. Truth is, pictures (and video) from the camera capture API are just stored in a temp file; if you set a breakpoint in the debugger and look at `capturedFile`, you'll see that it has an ugly file path like `C:\Users\kraigb\AppData\Local\Packages\ ProgrammingWin8-JS-CH3-HereMyAm3a_5xchamk3agtd6\TempState\picture001.png`. Egads. Not the friendliest of locations, and definitely not one that we'd want a typical consumer to ever see!

With an app like this, let's copy that temp file to a more manageable location, which could, for example, allow the user to select from previously captured pictures. We'll make a copy in the app's local appdata folder and use `ms-appdata` to set the `img src` to that location. Let's start with the call to `captureUI.captureFileAsync` as before:

```
//For use across chained promises
var capturedFile = null;

captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
  .then(function (capturedFileTemp) {
    //Be sure to check validity of the item returned; could be null if the user canceled.
    if (!capturedFileTemp) { throw ("no file captured"); }
  });
```

Notice that instead of calling `done` to get the results of the promise, we're using `then` instead. This is because we need to chain a number of async operations together and `then` allows errors to propagate through the chain, as we'll see in the next section. In any case, once we get a result in `capturedFileTemp` (which is in a gnarly-looking folder), we then open or create a "HereMyAm" folder within our local appdata. This happens via `Windows.Storage.ApplicationData.current.localFolder`, which gives us a `Windows.Storage.StorageFolder` object that provides a `createFolderAsync` method:

```
//As a demonstration of ms-appdata usage, copy the StorageFile to a folder called HereMyAm
//in the appdata/local folder, and use ms-appdata to point to that.
var local = Windows.Storage.ApplicationData.current.localFolder;
capturedFile = capturedFileTemp;
return local.createFolderAsync("HereMyAm",
  Windows.Storage.CreationCollisionOption.openIfExists);
})
.then(function (myFolder) {
  //Again, check validity of the result operations
```

```
if (!myFolder) { throw ("could not create local appdata folder"); }
```

Assuming the folder is created successfully, `myFolder` will contain another `StorageFile` object. We then use this as a target parameter for the temp file's `copyAsync` method, which also takes a new filename as its second parameter. For that name we'll just use the original name with the date/time appended (replacing colons with hyphens to make a valid filename):

```
//Append file creation time (should avoid collisions, but need to convert colons)
var newName = capturedFile.displayName + " - "
    + capturedFile.dateCreated.toString().replace(/:/g, "-") + capturedFile.fileType;
return capturedFile.copyAsync(myFolder, newName);
})
.done(function (newFile) {
    if (!newFile) { throw ("could not copy file"); }
```

Because this was the last async operating in the chain, we use the promise's `done` method for reasons we'll again see in a moment. In any case, if the copy succeeded, `newFile` contains a `StorageFile` object for the copy, and we can point to that using an `ms-appdata` URI:

```
lastCapture = newFile; //Save for Share
that.src = "ms-appdata:///local/HereMyAm/" + newFile.name;
},
function (error) {
    console.log(error.message);
});
```

The completed code is in the `HereMyAm3a` example.

Of course, we could still use `URL.createObjectURL` with `newFile` as before (making sure to provide the `{ oneTimeOnly=true }` parameter to avoid memory leaks). While that would defeat the purpose of this exercise, it works perfectly (and the memory overhead is essentially the same since the picture has to be loaded either way). In fact, we'd need to use it if we copy images to the user's pictures library instead. To do this, just replace `Windows.Storage.ApplicationData.current.localFolder` with `Windows.Storage.KnownFolders.picturesLibrary` and declare the Pictures library capability in the manifest. Both APIs give us a `StorageFolder`, so the rest of the code is the same except that we'd use `URL.createObjectURL` because we can neither use `ms-appdata://` nor `file://` to refer to the pictures library. The `HereMyAm3a` example contains this code in comments.

Sequential Async Operations: Chaining Promises

In the previous code example, you might have noticed how we throw exceptions whenever we don't get a good result back from any given async operation. Furthermore, we have only a single error handler at the end, and there's this odd construct of returning the result (a promise) from each subsequent async operation instead of just processing the promise then and there.

Though it may look odd at first, this is actually the most common pattern for dealing with sequential async operations because it works better than the more obvious approach of nesting.

Nesting means to call the next async API within the completed handler of the previous one, with each promise fulfilled with `done`. Here's how the async calls in previous code would be placed with this approach (extraneous code removed for simplicity):

```
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    .done(function (capturedFileTemp) {
        //...
        local.createFolderAsync("HereMyAm", ...)
            .done(function (myFolder) {
                //...
                capturedFile.copyAsync(myFolder, newName)
                    .done(function (newFile) {
                        })
                })
        });
```

The one advantage to this approach is that each completed handler will have access to all the variables declared before it. Yet the disadvantages begin to pile up. For one, there is usually enough intervening code between the async calls that the overall structure becomes visually messy. More significantly, error handling becomes significantly more difficult. When promises are nested, error handling must be done at each level; if you throw an exception at the innermost level, for instance, it won't be picked up by any of the outer error handlers. Each promise thus needs its own error handler, making real spaghetti of the basic code structure:

```
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    .done(function (capturedFileTemp) {
        //...
        local.createFolderAsync("HereMyAm", ...)
            .done(function (myFolder) {
                //...
                capturedFile.copyAsync(myFolder, newName)
                    .done(function (newFile) {
                        },
                        function (error) {
                            })
                },
                function (error) {
                    });
        },
        function (error) {
            });
```

I don't know about you, but I really get lost in all the }'s and)'s (unless I try hard to remember my LISP class in college), and it's hard to see which error function applies to which async call.

Chaining promises solves all of this with the small tradeoff of needing to declare a few extra temp variables outside the chain. With chaining, you `return` the promise out of each completed handler (rather than calling the next async function and tagging on a `.done`). This allows you to indent all the async calls at the same level, and it also has the effect of propagating errors down the chain. When an error happens within a promise, you see, what comes back is still a promise object, and if you call its `then` method (but not `done`—see the next section), it will again return *another* promise object with an

error. As a result, any error along the chain will quickly propagate through to the first available error handler, thereby allowing you to have only a single error handler at the end:

```
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    .then(function (capturedFileTemp) {
        //...
        return local.createFolderAsync("HereMyAm", ...);
    })
    .then(function (myFolder) {
        //...
        return capturedFile.copyAsync(myFolder, newName);
    })
    .done(function (newFile) {
        },
        function (error) {
        })
    })
```

To my eyes (and my aging brain), this is a much cleaner code structure—and it's therefore easier to debug and maintain. If you like, you can even end the chain with a `done(null, errorHandler)` call, replacing the previous `done` with `then`:

```
captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    //...
    .then(function (newFile) {
    })
    .done(null, function (error) {
    })
})
```

Finally, a word about debugging chained promises (or nested ones, for that matter). Each step involves an async operation, so you can't just step through as you would with synchronous code (otherwise you'll end up deep inside WinJS). Instead, set a breakpoint on the first line within each completed handler and on the first line of the error function at the end. As each breakpoint is hit, you can step through that completed handler. When you reach the next async call, click the Continue button in Visual Studio so that the async operation can run, after which you'll hit the breakpoint in the next completed handler or you'll hit the breakpoint in the error handler.

Error Handling Within Promises: then vs. done

Although it's common to handle errors at the end of a chain of promises, as demonstrated in the code above, you can still provide an error handler at any point in the chain—`then` and `done` both take the same arguments. If an exception occurs at that level, it will surface in the innermost error handler.

This brings us to the difference between `then` and `done`. First, `then` returns another promise, thereby allowing chaining, whereas `done` returns `undefined` (so it's always at the end of the chain). Second, if an exception occurs within one async operation's `then` method and there's no error handler at that level, the error gets stored in the promise returned by `then`. In contrast, if `done` sees an exception and there's no error handler, it throws that exception to the app's event loop. This will bypass any local (synchronous) `try/catch` block, though you can pick them up in either in the `WinJS.Application.-`

`onerror` or `window.onerror` events. (The latter will get the error if the former doesn't handle it.) If you don't, the app will be instantly terminated and an error report sent to the Windows Store dashboard. We actually recommend that you provide a `WinJS.Application.onerror` handler for this reason.

In practical terms, this means that if you end a chain of promises with a `then` and not `done`, all exceptions in that chain will get swallowed and you'll never know there was a problem! This can place an app in an indeterminate state and cause much larger problems later on. So, unless you're going to pass the last promise in a chain to another piece of code that will itself call `done`, always use `done` at the end of a chain even for a single async operation.

There is much more you can do with promises, by the way, like combining them, canceling them, and so forth. We'll come back to all this at the end of this chapter.

Debug Output, Error Reports, and the Event Viewer

Speaking of exceptions and error handling, it's sometimes heartbreaking to developers that `window.prompt` and `window.alert` are not available to WinRT apps as quickie debugging aids. Fortunately, you have two other good options for that purpose. One is `Windows.UI.Popups.-MessageDialog`, which is actually what you use for real user prompts in general. The other is to use `console.log`, as shown earlier, which will send text to Visual Studio's output pane. These messages can also be logged as Windows events, as we'll see in a moment.¹⁸

Another DOM API function to which you might be accustomed is `window.close`. You can still use this as a development tool, but Windows interprets this call in released apps as a crash and generates an error report in response. This report will appear in the Store dashboard for your app, with a message telling you to not use it! (After all, WinRT apps should not provide their own close affordances.)

There might be situations, however, when a released app needs to close itself in response to unrecoverable conditions. Although you can use `window.close` for this, it's better to use `MSApp.terminateApp` because it allows you to also include information as to the exact nature of the error that shows up in the Store dashboard, making it easier to diagnose the problem.

In addition to the Store dashboard, you should make fast friends with the Windows Event Viewer.¹⁹ This is where error reports, console logging, and unhandled exceptions (which again terminate the app without warning) can be recorded.

To enable this, you need to do a couple steps. First navigate to Application and Services Log and expand Microsoft/Windows/AppHost, left-click select (this is important), right-click Admin, and then select View -> Show Analytic and Debug Logs for full output, as shown in Figure 3-4. This will enable

¹⁸ For readers who are seriously into logging, beyond the kind you do with chainsaws, check out the [WinJS.Utilities](#) functions [startLog](#), [stopLog](#), and [formatLog](#), which provide additional functionality on top of `console.log`. I'll leave you to commune with the documentation for these but wanted to bring them to your awareness.

¹⁹ If you can't find Event Viewer, press the Windows key to go to the Start page, and then invoke the Settings charm. Select Tiles, and turn on Show Administrative Tools. You'll then see a tile for Event Viewer on your Start page.

tracing for errors and exceptions. Then right-click AppTracing (also under AppHost) and select Enable Log. This will trace your calls to `console.log` as well as other diagnostic information coming from the app host.

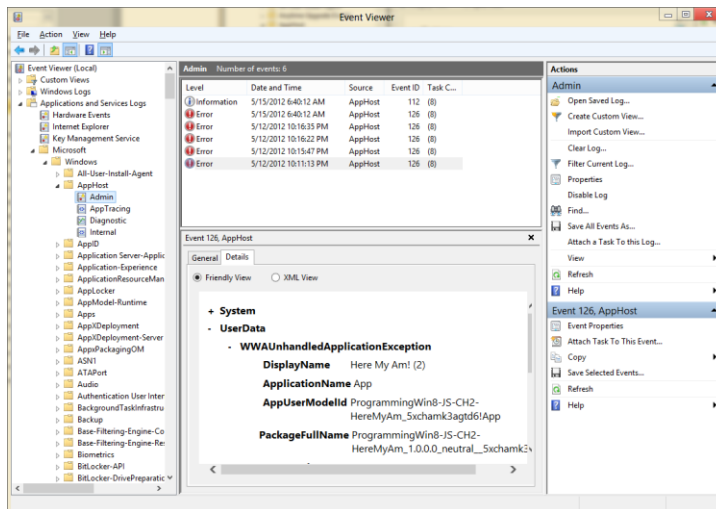


Figure 3-4 App host events, such as unhandled exceptions and load errors, can be found in Event Viewer.

We already introduced the Visual Studio's Exceptions dialog in Chapter 2; refer back to Figure 2-16. For each type of JavaScript exception, this dialog supplies two checkboxes labeled Thrown and User-unhandled. Checking Thrown will display a dialog box in the debugger (Figure 3-5) whenever an exception is thrown, regardless of whether it's handled and before reaching any of your error handlers. If you have error handlers, you can safely click the Continue button in the dialog, and you'll eventually see the exception surface in those error handlers. (Otherwise the app will terminate.) If you click Break instead, you can find the exception details in the debugger's Locals pane, as shown in Figure 3-6.

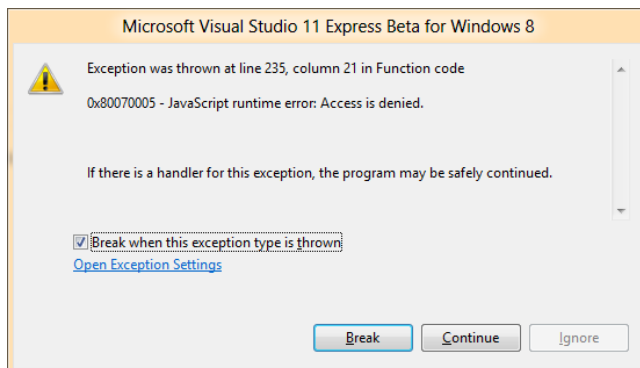


Figure 3-5 Visual Studio's exception dialog. As the dialog indicates, it's safe to press Continue if you have an error handler in the app; otherwise the app will terminate. Note that the checkbox in this dialog is a shortcut to toggle the Thrown checkbox for this exception type in the Exceptions dialog.

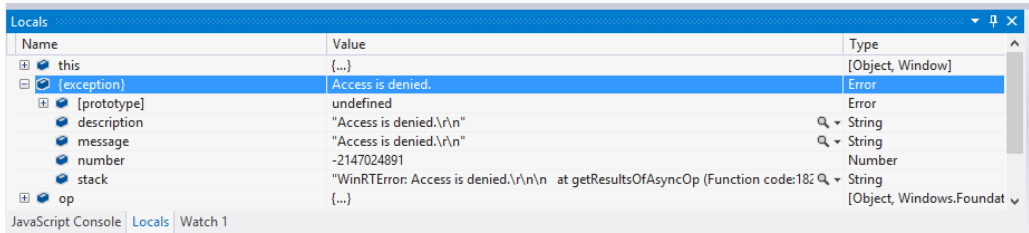


Figure 3-6 Information in Visual Studio's Locals pane when you Break on an exception.

The User-unhandled option (enabled for all exceptions by default) will display a similar dialog whenever an exception is thrown to the event loop, indicating that it wasn't handled by an app-provided error function ("user" code from the system's perspective).

You typically turn on Thrown only for those exceptions you care about; turning them all on can make it very difficult to step through your app! Still, you can try it as a test, and then leave checks only for those exceptions you expect to catch. Do leave User-unhandled checked for everything else; in fact, unless you have a specific reason not to, make sure that User-unhandled is checked next to JavaScript Runtime Exceptions because this will include those exceptions not otherwise listed. This way you can catch (and fix) any exceptions that might abruptly terminate the app, which is something your customers should never experience.

App Activation

First, let me congratulate you for coming this far into a very detailed chapter! As a reward, let's talk about something much more tangible and engaging: the actual activation of an app and its startup sequence, something that can happen a variety of ways, such as via the Start screen tile, contracts, and file type and protocol associations. In all these activation cases, you'll be writing plenty of code to initialize your data structures, reload previously saved state, and do everything to establish a great experience for your users.

Branding Your App 101: The Splash Screen and Other Visuals

With activation, we actually need to take a step back even *before* the app host gets loaded, back to the moment a user taps your tile on the Start screen or when your app is launched through a contract or other association. The very first thing that happens, before any app-specific code is loaded or run, is that Windows displays a splash screen composed of the image and background color you provide in your manifest.

The splash screen—which shows for at least 0.75 seconds so that it's not just a flash—gives users something interesting to look at briefly while the app gets started (much better than an hourglass). It also occupies the whole view where the app is being launched (which might be the filled view state or the overlay area from the share or search charm), so it's a much more directly engaging experience for

your users. During this time, an instance of the app host gets launched to load, parse, and render your HTML/CSS, and load, parse, and execute your JavaScript, firing events along the way as we'll see in the next section. When the app is ready with its first page, the system removes the splash screen.

The splash screen, along with your app tile, is clearly one of the most important ways to uniquely brand your app, so make sure that you and your graphic artist(s) give full attention to these. There are additional graphics and settings in the manifest that also affect your branding and overall presence in the system, as shown in the table below. Be especially aware that the Visual Studio and Blend templates provide some default and thoroughly unattractive placeholder graphics. Thus, take a solemn vow right now that you truly, truly, cross-your-heart will *not* upload an app to the Windows Store with those defaults still in place! (For additional guidance, see [Guidelines and checklist for splash screens.](#))

You can see that the table lists multiple sizes for various images specified in the manifest to accommodate varying pixel densities: 100%, 140%, and 180% scale factors, and even a few at 80% (don't neglect the latter: they are typically used for most desktop monitors). So while you can just provide a single 100% scale image for each of these, it's almost guaranteed that scaled-up versions of that graphic are going to look bad. So why not make your app look its best? Take the time to create each individual graphic consciously.

Manifest Tab	Section	Item	Use	Image Sizes 100%	140%	180%
Packaging	n/a	Logo	Tile/logo image used for the app on its Product Description Page in the Windows Store.	50x50	70x70	90x90
Application UI	n/a	Display Name	Appears in "all apps" view on the Start screen, search results, the Settings charm, and in the Store.	n/a	n/a	n/a
	Tile	Logo	Single-wide tile image	150x150 (+80% scale at 120x120)	210x210	270x270
		Wide logo (Optional)	Double-wide tile image. If provided, this is shown as the default, but user can use the single-wide tile if desired.	310x150 (+80% scale at 248x120)	434x210	558x270
		Small logo	Tile used in zoomed-out and "all apps" views of the Start screen, and in the Search and Share panes if the app supports those contracts. Also used on the app tile if you elect to show a logo instead of the app name in the lower left cover of the tile.	30x30 (+80% scale at 24x24)	42x42	54x54
		Show name	Specifies whether to show the app name on your app tile (both, neither, or the single- or double-wide specifically). Set this to "no logo" if your tile images includes your app name.	n/a	n/a	n/a

		Short name	Optional: if provided, is used for the name on the tile in place of the Display Name, as Display Name may be too long for a single-wide tile	n/a	n/a	n/a
		Foreground text	Color of name text shown on the tile if applicable (see Show name). Options are Light and Dark. There must be a 1.5 contrast ration between this and the background color	n/a	n/a	n/a
		Background color	Color that will be shown for transparent areas of any tile images, buttons in app dialogs, notification backgrounds, and a few other places. Also provides the splash screen background color unless that is set separately.	n/a	n/a	n/a
	Notifications	Badge logo	Shown next to a badge notification to identify the app on the lock screen (uncommon, as this requires additional capabilities to be declared).	24x24	33x33	43x43
	Splash screen	Splash screen	When the app is launched, this image is shown in the center of the screen against the Background color. The image can utilize transparency if designed.	620x300	868x420	1116x540
		Background color	Color that will fill the majority of the splash screen; if not set, the App UI Background color is used.	n/a	n/a	n/a

In the table, note that 80% scale tile graphics are used in specific cases like low DPI modes and should be provided with other scaled images. Note also that there are additional graphics besides the Packaging Logo (first item in the table) that you'll need when uploading an app to the Windows Store. See the [App images](#) topic in the docs under "Promotional images" for full details.

When saving these files, append *.scale-80*, *.scale-100*, *.scale-140*, and *.scale-180* to the filenames, before the file extension, as in *splashscreen.scale-140.png*. This allows you, both in the manifest and elsewhere in the app, to refer to an image with just the base name, such as *splashscreen.png*, and Windows will automatically load the appropriate scaled variant. Otherwise it looks for one without the suffix. No code needed! This is demonstrated in the HereMyAm3b example, where I've added all the various branded graphics (with some additional text in each graphic to show the scale). To test these different graphics, use the set resolution/scaling button in the simulator—refer back to Figure 2-5—to choose different pixel densities on a 10.6" screen (1366 x 768 = 100%, 1920 x 1080 = 140%, and 2560 x 1440 = 180%). You'll also see the 80% scale used on the other display choices, including the 23" and 27" settings. In all cases, the setting affects which images are used on the Start screen and the splash screen, but note that you might need to exit and restart the simulator to see the new scaling take effect.

One thing you might also notice is that full-color photographic images, as I'm using in HereMyAm3b here, don't scale down very well to the smallest sizes (Store logo and small logo). This is one reason why such logos are typically simpler with WinRT app design, so hopefully your designers do

a better job than I have!

Activation Event Sequence

As the app host is built on the same parsing and rendering engines as Internet Explorer, the general sequence of activation events is more or less what a web application sees in a browser. Actually, it's more rather than less! When you launch an app from its tile, here's the process as Windows sees it:

1. Windows displays a splash screen using information from the app manifest.
2. Windows launches the app host, identifying the app to launch.
3. The app host retrieves the app's Start Page setting (see the Application UI tab in the manifest editor), which identifies the HTML page to load.
4. The app host loads that page along with referenced stylesheets and script (deferring script loading if indicated in the markup). Here it's important that all files are properly encoded for best startup performance. (See the sidebar below.)
5. `document.DOMContentLoaded` fires. You can use this to do further initialization specifically related to the DOM, if desired (not common).
6. `Windows.UI.WebUI.WebUIApplication.onactivated` fires. This is typically where you'll do all your startup work, instantiate WinJS and custom controls, initialize state, and so on.
7. The splash screen is hidden once the activated event handler returns (unless the app has requested a deferral as discussed later in the "Activation Deferrals" section).
8. `body.onload` fires. This is typically not used in WinRT apps, though it might be utilized by imported code or third party libraries.

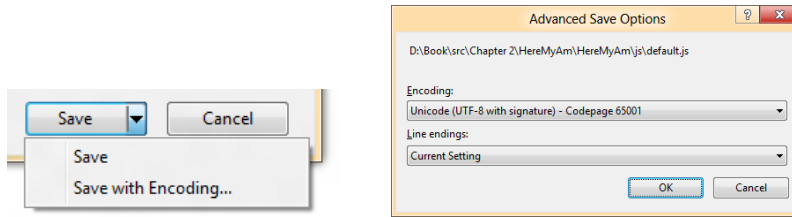
What's also very different is that an app can again be activated for many different purposes, such as contracts and associations, even while it's already running. As we'll see in later chapters, the specific page that gets loaded (step 3) can vary by contract, and if a particular page is already running, it will receive only the `Windows.UI.WebUI.WebUIApplication.onactivated` event and not the others.

For the time being, though, let's concentrate on how we work with this core launch process, and because you'll generally do your initialization work within the `activated` event, let's examine that structure more closely.

Sidebar: File Encoding for Best Startup Performance

To optimize bytecode generation when parsing HTML, CSS, and JavaScript, the Windows Store requires that all .html, .css, and .js files are saved with Unicode UTF-8 encoding. This is the default for all files created in Visual Studio or Blend. If you're importing assets from other sources, check this encoding: in Visual Studio's File Save As dialog (Blend doesn't have this at present), select *Save with Encoding* and set that to *Unicode (UTF-8 with signature) – Codepage 65001*. The

Windows App Certification Kit will issue warnings if it encounters files without this encoding.



Along these same lines, minification of JavaScript isn't particularly important for WinRT apps. Because an app package is downloaded from the Windows Store as a unit and often contains other assets that are much larger than your code files, minification won't make much difference there. Once the package is installed, bytecode generation means that the package's JavaScript has already been processed and optimized, so minification won't have any additional performance impact.

Activation Code Paths

As we saw in Chapter 2, new projects created in Visual Studio or Blend give you the following code (with a few more comments) in `default.js`:

```
(function () {
    "use strict";

    var app = WinJS.Application;
    var activation = Windows.ApplicationModel.Activation;

    app.onactivated = function (args) {
        if (args.detail.kind === activation.ActivationKind.launch) {
            if (args.detail.previousExecutionState !==
                activation.ApplicationExecutionState.terminated) {
                // TODO: This application has been newly launched. Initialize
                // your application here.
            } else {
                // TODO: This application has been reactivated from suspension.
                // Restore application state here.
            }
            args.setPromise(WinJS.UI.processAll());
        }
    };

    app.oncheckpoint = function (args) {
    };

    app.start();
})();
```

Let's go through this piece by piece to review what we already learned and complete our understanding of this essential code structure:

- `(function () { ... })()`; is again the JavaScript module pattern.
- `"use strict"` instructs the JavaScript interpreter to apply [Strict Mode](#), a feature of ECMAScript 5. This checks for sloppy programming practices (like using implicitly declared variables), so it's a good idea to leave it in place.
- `var app = WinJS.Application;` and `var activation = Windows.Application-Mode.Activation;` both create substantially shortened aliases for commonly used fully qualified namespaces. This is a common practice to simplify multiple references to the same part of WinJS or WinRT.
- `app.onactivated = function (args) {...}` assigns a handler for the `WinJS.UI.onactivated` event, which is a wrapper for `Windows.UI.WebUI.WebUIApplication.onactivated`. In this handler:
 - `args.detail.kind` identifies the type of activation.
 - `args.detail.previousExecutionState` identifies the state of the app prior to this activation, which determines whether to reload state.
 - `WinJS.UI.processAll` instantiates WinJS controls—that is, elements that contain a `data-win-control` attribute, as we'll cover in Chapter 4.
 - `args.setPromise` instructs Windows to wait until `WinJS.UI.processAll` is complete before removing the splash screen. (See "Activation Deferrals" later in this chapter.)
 - `app.oncheckpoint` gets an empty handler in the template; we'll cover this in the "App Lifecycle Transition Events" section later in this chapter.
 - `app.start()` (`WinJS.Application.start()`) initiates processing of events that WinJS queues during startup.

Notice how we're not directly handling any of the events that Windows is firing, like `DOMContentLoaded` or `Windows.UI.WebUI.WebUIApplication.onactivated`. Are we just ignoring those events? Not at all: one of the convenient services that WinJS offers, through `WinJS.UI.Application`, is a simplified structure for activation and other app lifetime events. Entirely optional, but very helpful.

With `start`, for example, a couple of things are happening. First, the `WinJS.Application` object listens for a variety of events that come from different sources (the DOM, WinRT, etc.) and coalesces them into a single object with which you register your own handlers. Second, when `WinJS.Application` receives activation events, it doesn't just pass them on to the app's handlers, because your handlers might not, in fact, have been set up yet. So it queues those events until the app says it's really ready by calling `start`. At that point WinJS goes through the queue and fires those events. That's really all there is to it.

As the template code shows, apps typically do most of their initialization work within the `activated`

event, but there are a number of potential code paths depending on the values in `args.details` (an [IActivatedEventArgs](#) object). If you look at the documentation for [WinJS.Application.onactivated](#), you'll see that the exact contents of `args.details` depends on specific *kind* of activation. All activations, however, share three common properties:

args.details Property	Type (in Windows.Application- Model.Activation)	Description
Kind	ActivationKind	The reason for the activation. The possibilities are launch (most common); search , shareTarget , file , protocol , fileOpenPicker , fileSavePicker , contactPicker , and cachedFileUpdater (for servicing contracts); and device , printTaskSettings , and cameraSettings (generally used with device apps). For each supported activation kind, the app will have an appropriate initialization path.
previousExecutionState	ApplicationExecutionState	The state of the app prior to this activation. Values are notRunning , running , suspended , terminated , and closedByUser . Handling the terminated case is most common because that's the one where you want to restore previously saved state (see "App Lifecycle Transition Events").
splashScreen	SplashScreen	Contains an ondismissed property to assign a handler that to perform other actions when the system splash screen is dismissed. This also contains an imageLocation property (Windows.Foundation.Rect) with coordinates where the splash screen image was displayed, as noted in "Extended Splash Screens."

Additional properties provide relevant data for the activation. For example, [launch](#) provides the [tileId](#) and [arguments](#), which are needed with secondary tiles. (See Chapter 13, "Tiles, Notifications, the Lock Screen, and Background Tasks"). The [search](#) kind (the next most commonly used) provides [queryText](#) and [language](#), [protocol](#) provides a [uri](#), and so on. We'll see how to use many of these in the proper context, and sometimes they apply to altogether different pages than `default.html`. What's contained in the templates (and what we've already used for an app like Here My Am!) is primarily to handle normal startup from the app tile (or within Visual Studio's debugger).

WinJS.Application Events

[WinJS.Application](#) isn't concerned only with activation—its purpose is to centralize events from several different sources and turn them into events of its own. Again, this enables the app to listen to events from a single source (either assigning handlers via [addEventListener\(<event>\)](#) or [on<event>](#) properties; both are supported). Here's the full rundown on those events and when they're fired (if queued, the event is fired within [WinJS.Application.start](#)):

- [activated](#) Queued in the local context for [Windows.UI.WebUI.WebUIApplication.onactivated](#). In the web context, where WinRT is not applicable, this is instead queued for [DOMContentLoaded](#) (where the launch kind will be [launch](#) and

`previousExecutionState` is set to `notRunning`).

- `loaded` Queued for `DOMContentLoaded` in all contexts;²⁰ in the web context, will be queued prior to `activated`.
- `ready` Queued after `loaded` and `activated`. This is the last one in the activation sequence.
- `error` Fired if there's an exception in dispatching another event. (If the error is not handled here, it's passed onto `window.onerror`.)
- `checkpoint` This tells the app when to save the state it needs to restart from a previous state of `terminated`. It's fired in response to both the document's `beforeunload` event, as well as `Windows.UI.WebUI.WebUIApplication.onsuspending`.
- `unload` Also fired for `beforeunload` after the `checkpoint` event is fired.
- `settings` Fired in response to `Windows.UI.ApplicationSettings.SettingsPane.-oncommandsrequested`. (See Chapter 8, "State, Settings, Files, and Documents.")

With most of these events (except `error` and `settings`), the `args` you receive contains a method called `setPromise`. If you need to perform an async operation within an event handler (like an `XmlHttpRequest`), you can obtain the promise for that work and hand it off to `setPromise` instead of calling its `then` or `done` yourself. WinJS will then *not* process the next event in the queue until that promise is fulfilled. Now to be honest, there's no actual difference between this and just calling `done` on the promise yourself within the `loaded`, `ready`, and `unload` events. It *does* make a difference with `activated` and `checkpoint` (specifically the suspending case) because Windows will otherwise assume that you've done everything you need as soon as you return from the handler; more on this in the "Activation Deferrals" section. So, in general, if you have async work within these events handlers, it's a good habit to use `setPromise`. Because `WinJS.UI.processAll` is itself an async operation, the templates wrap it with `setPromise` so that the splash screen isn't removed until WinJS controls have been fully instantiated.

Anyway, I think you'll generally find `WinJS.Application` to be a useful tool in your apps, and it also provides a few more features as documented on the [WinJS.Application page](#). For example, it provides `local`, `temp`, `roaming`, and `sessionState` properties, which are helpful for managing state, as we'll see later on in this chapter and in Chapter 8.

The other bits are the `queueEvent` and `stop` methods. The `queueEvent` method drops an event into the queue that will get dispatched (after any existing queue is clear) to whatever listeners you've set up on the `WinJS.Application` object. Events are simply identified with a string, so you can queue an event with any name you like, and call `WinJS.Application.addEventListener` with that same name

²⁰ There is also the `WinJS.Utilities.ready` API through which you can specifically set a callback that's called for `DOMContentLoaded`. This is used within WinJS, in fact, to guarantee that any call to `WinJS.UI.processAll` is processed after `DOMContentLoaded`.

anywhere else in the app. This can be useful for centralizing custom events that you might invoke both during startup and at other points during execution without creating a separate global function for that purpose. It's also a powerful means through which separately defined, independent components can raise events that get aggregated into a single handler.

As for `stop`, this is provided to help with unit testing so that you can simulate different activation sequences without having to relaunch the app and somehow simulate the right conditions when it restarts. When you call `stop`, WinJS removes its listeners, clears any existing event queue, and clears the `sessionState` object, but the app continues to run. You can then call `queueEvent` to populate the queue with whatever events you like and then call `start` again to process that queue. This process can be repeated as many times as needed.

Extended Splash Screens

Now, though the default splash screen helps keep the user engaged, they won't stay engaged if that same splash screen stays up for a really long time. In fact, "a really long time" for the typical consumer amounts to all of 15 seconds, at which point they'll pretty much start to assume that the app has hung and return to the Start screen to launch some other app that won't waste their afternoon.

In truth, so long as the user keeps your app in the foreground and doesn't switch away, Windows will give you all the time you need. But if the user switches to the Start screen or another app, you're subject to a 15-second timeout. If you're not in the foreground, Windows will wait only 15 seconds for an app to get through `app.start` and the `activated` event, at which point your home page should be rendered. Otherwise, boom! Windows automatically terminates your app.

The first consideration, of course, is to optimize your startup process to be as quick as possible. Still, sometimes an app really needs more than 15 seconds to get going, especially on its first run, so it should let the user know that something is happening. For example, an app package might include a bunch of compressed data when downloaded from the Store, which it needs to expand onto the local file system on first run so that subsequent launches are much faster. Many games do this with graphics and other resources (optimizing the local storage for device characteristics); other apps might to populate a local IndexedDB from data in a JSON file or download and cache a bunch of data from an online service.

It's also possible that the user is trying to launch your app shortly after rebooting the system, in which case there might be lots of disk activity going on. If you load data from disk in your activation path, your process could take much longer than usual. Such apps thus implement an *extended splash screen*, which is just a fancy term for some clever fakery. Simply said, if the app determines that it needs more time, it hides its real home page behind another `div` that looks exactly like the system-provided splash screen but that is under the app's control so that it can display progress indicators or other custom UI while initialization continues.

In general, Microsoft recommends that the extended splash screen initially matches the system splash screen to avoid visual jumps. (See [Guidelines and checklist for splash screens](#).) At this point many

apps simply add a progress indicator with some kind of a “Please go grab a drink, do some jumping jacks, or enjoy a few minutes of meditation while we load everything” message. Matching the system splash screen, however, doesn’t mean that the extended splash screen has to stay that way. A number of apps start with a replica of the system splash screen and then animate the graphic to one side to make room for other elements. Other apps fade out the initial graphic and start a video.

Making a smooth transition is the purpose of the `args.detail.splashScreen` object included with the `activated` event. This object—see [Windows.ApplicationModel.Activation.-SplashScreen](#)—contains an `imageLocation` property, which is a `Windows.Foundation.Rect` containing the placement and size of the splash screen image. Because your app can be run on a variety of different display sizes, this tells you where to place the same image on your own page, where to start an animation, and/or where to place things like messages and progress indicators relative to that image.

The `splashScreen` object also provides an `ondismissed` event so that you can perform specific actions when the system-provided splash screen is dismissed and your first page comes up. Typically, this is useful to trigger the start of on-page animations, starting video playback, and so on.

We won’t have a need to implement an extended splash screen in this chapter’s examples, but you can refer to the [Splash Screen sample](#) in the SDK. One more detail that’s worth mentioning is that because an extended splash screen is just a page in your app, it can be placed into the various view states such as snap view. So, as with every other page in your app, make sure your extended splash screen handles those states!

Activation Deferrals

As mentioned earlier, once you return from the `activated` event, Windows assumes that you’ve done everything you need on startup. By default, then, Windows will remove its splash screen and make your home page visible. But what if you need to complete one or more async operations before that home page is really ready, such as completing `WinJS.UI.processAll`?

This, again, is what the `args.setPromise` method inside the `activated` event is for. If you give your async operation’s promise to `setPromise`, Windows will wait until that promise is fulfilled before taking down the splash screen. The templates use this to keep the system splash screen up until `WinJS.UI.processAll` is complete.

As `setPromise` just waits for a single promise to complete, how do you handle multiple async operations? You can do this a couple of ways. First, if you need to control the sequencing of those operations, you can chain them together as we already know how to do—just be sure that the end of the chain is a promise that becomes the argument to `setPromise`—don’t call its `done` method (use `then` if needed)! If the sequence isn’t important but you need all of them to complete, you can combine those promises by using `WinJS.Promise.join`, passing the result to `setPromise`. If you need only one of the operations to complete, you can use `WinJS.Promise.any` instead—`join` and `any` are discussed in the last section of this chapter.

The other means is to register more than one handler with `WinJS.Application.onactivated`; each handler will get its own event args and its own `setPromise` function, and WinJS will combine those returned promises together with `WinJS.Promise.join`.

Now the `setPromise` method coming from WinJS is actually implemented using a more generic deferral mechanism from WinRT. The `args` given to `Windows.UI.WebUI.WebUIApplication.-onactivated` (the WinRT event) contains a little method called `getDeferral` (technically `Windows.UI.WebUI.ActivatedOperation.getDeferral`). This function returns a deferral object that then contains a `complete` method, and Windows will leave the system splash screen up until you call that method (although this doesn't change the fact that users are impatient and your app is still subject to the 15-second limit!). The code looks like this:

```
//In the activated handler
var activatedDeferral = Windows.UI.WebUI.ActivatedOperation.getDeferral();

//After initialization is complete
activatedDeferral.complete();
```

Of course, `setPromise` ultimately does exactly this, and if you add a handler for the WinRT event directly, you can use the deferral yourself.

App Lifecycle Transition Events and Session State

To an app—and the app's publisher—a perfect world might be one in which consumers ran that app and stayed in that app forever (making many in-app purchases, no doubt!). Well, the hard reality is that this just isn't reality. No matter how much you'd love it to be otherwise, yours is not the only app that the user will ever run. After all, what would be the point of features like sharing or snapping if you couldn't have multiple apps running together? For better or for worse, users will be switching between apps, changing view states, and possibly closing your app. But what you *can* do is give energy to the "better" side of the equation by making sure your app behaves well under all these circumstances.

The first consideration is *focus*, which applies to controls in your app as well as to the app itself. Here you can simply use the standard HTML `blur` and `focus` events. For example, an active game or one with a timer would typically pause itself on `blur` and perhaps restart again on `focus`.

A similar but different condition is *visibility*. An app can be visible but not have the focus, as when it's snapped. In such cases an app would continue things like animations or updating a feed, but it would stop such activities when visibility is lost (that is, when the app is actually in the background). For this, use the [visibilitychange event](#) in the DOM API, and then examine the [visibilityState property](#) of the `window` or `document` object, as well as the `document.hidden` property. (The event works for visibility of all other elements as well.) A change in visibility is also a good time to save user data like documents or game progress.

For *view state changes*, an app can pick these up in several ways. As shown in the Here My Am! example, an app typically uses media queries (in declarative CSS or in code through media query

listeners) to reconfigure layout and visibility of elements, which is really all that view states should affect. (Again, view state changes never change the mode of the app, just layout and object visibility.) At any time, an app can also retrieve the current view state through `Windows.UI.ViewManagement.ApplicationView.value`. This returns one of the `Windows.UI.ViewManagement.ApplicationViewState` values: `snapped`, `filled`, `fullScreenLandscape`, and `fullScreenPortrait`; details in Chapter 6, “Layout.”

When your app is closed (the user swipes top to bottom or presses Alt+F4), it’s important to note that the app is first moved off-screen (hidden), suspended, and then terminated, so the typical DOM events like `unload` aren’t much use. A user might also kill your app in Task Manager, but this won’t generate any events in your code either. Remember also that apps should *not* close themselves, as discussed before, but they can use `MSApp.terminateApp` to close due to unrecoverable conditions.

Suspend, Resume, and Terminate

Beyond focus, visibility, and view states, three other critical moments in an app’s lifetime exist:

- **Suspending** When an app is not visible (in any view state), it will be suspended after five seconds (according to the wall clock) to conserve battery power. This means it remains wholly in memory but won’t be scheduled for CPU time and thus won’t have network or disk activity (except when using specifically allowed background tasks). When this happens, the app receives the `Windows.UI.WebUI.WebUIApplication.onsuspending` event, which is also exposed through `WinJS.Application.oncheckpoint`. Apps must return from this event within the five-second period, or Windows will assume the app is hung and terminate it (period!). During this time, apps save transient session state and should also release any exclusive resources acquired as well, like file handles or device access. (See [How to suspend an app](#).)
- **Resuming** If the user switches back to a suspended app, it receives the `Windows.UI.WebUI.WebUIApplication.onresuming` event. (This is not surfaced through `WinJS.Application` because it’s not commonly used and WinJS has no value to add.) We’ll talk more about this in the “Data from Services and WinJS.xhr” section coming up soon, because the need for this event often arises when using services. In addition, if you’re tracking a user’s location using the Geolocator, you won’t receive updates unless you’re using a background task (see Chapter 9, “Input and Sensors”), so you’ll want to refresh your location reading. There are also times when you might want to refresh your layout (as we’ll see in Chapter 6), because it is possible for your app to resume directly into a different view state than when it was suspended. The same goes for refreshing the state of clipboard commands (as we’ll see in Chapter 12, “Contracts”).
- **Terminating** When suspended, an app might be terminated if there’s a need for more memory. There is *no event* for this, because by definition the app is already suspended and no code can run.

It's very helpful to know that you can simulate these conditions in the Visual Studio debugger by using the toolbar drop-down shown in Figure 3-7. These commands will trigger the necessary events as well as set up the `previousExecutionState` value for the next launch of the app. (Be very grateful for these controls—there was a time when we didn't have them, and it was painful to debug these conditions!)

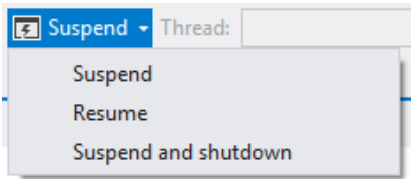


Figure 3-7 The Visual Studio toolbar drop-down to simulate suspend, resume, and terminate.

We've briefly listed those previous states before, but let's see how those relate to the events that get fired and the `previousExecutionState` value that shows up when the app is next launched. This can get a little tricky, so the transitions are illustrated in Figure 3-8 and the table below describes how the `previousExecutionState` values are determined.

Value of <code>previousExecutionState</code>	Scenarios
<code>notrunning</code>	First run after install from Store. First run after reboot or log off. App is launched within 10 seconds of being closed by user (about the time it takes to hide, suspend, and cleanly terminate the app; if the user relaunches quickly, Windows has to immediately terminate it without finishing the suspend operation). App was terminated in Task Manager while running or closes itself with <code>MSApp.terminateApp</code> .
<code>running</code>	App is <i>currently running</i> and then invoked in a way other than its app tile, such as Search, Share, secondary tiles, toast notifications, and all other contracts. When an app is running and the user taps the app tile, Windows just switches to the already-running app and without triggering activation events (though <code>focus</code> and <code>visibilitychange</code> will both be raised).
<code>suspended</code>	App is <i>suspended</i> and is invoked in a way other than the app tile (as above for <code>running</code>). In addition to focus/visibility events, the app will also receive the <code>resuming</code> event.
<code>terminated</code>	App was previously suspended and then terminated by Windows due to resource pressure. Note that this does not apply to <code>MSApp.terminateApp</code> because an app would have to be running to call that function.
<code>closedByUser</code>	App was closed by an uninterrupted close gesture (swipe down or Alt+F4). An "interrupted" close is when the user switches back to the app within 10 seconds, in which case the previous state will be <code>notrunning</code> instead.

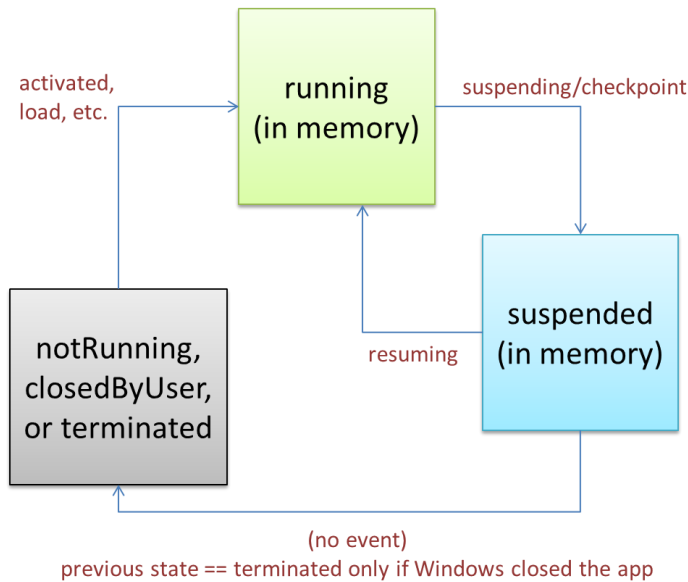


Figure 3-8 Process lifecycle events and previousExecutionState values.

The big question for the app, of course, is not so much what determines the value of `previousExecutionState` as what it should actually *do* with this value during activation. Fortunately, that story is a bit simpler and one that we’ve already seen in the template code:

- If the activation kind is `launch` and the previous state is `notrunning` or `closedByUser`, the app should start up with its default UI and apply any persistent settings (such as those in its Settings panel). With `closedByUser`, there might be scenarios where the app should perform additional actions (such as updating cached data) after the user explicitly closed the app and left it closed for a while.
- If the activation kind is `launch` and the previous state is `terminated`, the app should start up in the *same state* as when it was last suspended.
- For `launch` and other activation kinds that include additional arguments or parameters (as with secondary tiles, toast notifications, and contracts), it should initialize itself to serve that purpose by using the additional parameters. The app might already be running, so it won’t necessarily initialize its default state again.

The second requirement above is exactly why the templates provide a code structure for this case along with a `checkpoint` handler. We’ll see the full details of saving and reloading state in Chapter 8. The basic idea is that an app should, when being suspended, save whatever transient session state it would need to rehydrate itself after being terminated like form data, scroll positions, the navigation stack, and other variables. This is because although Windows might have suspended the app and dumped it from memory, *it’s still running in the mind of the user*. Thus, when users activate the app

again for normal use (activation kind is `launch`, rather than through a contract), they expect that app to be right where it was before. When an app gets suspended, it must save whatever state is necessary to make this possible, and it must restore that state when activated under these conditions. (For more on app design where this is concerned, see [Guidelines for app suspend and resume](#).) Be clear that if the user directly closes the app with Alt+F4 or the swipe-down gesture, the `suspending/checkpoint` events will also be raised, so the app still saves state. In these cases, however, the app will be automatically terminated after being suspended, and it won't be asked to reload that state when it's restarted because `previousExecutionState` will be `notRunning` or `closedByUser`.

The best practice is actually to save session state incrementally (as it changes) to minimize the work needed within the suspending event, because you have only five seconds to do it. Mind you, this session state does not include data that is persistent across sessions (like user files, high scores, and app settings) because an app would always reload or reapply such persistent data in each activation path. The only concern here is maintaining the illusion that the app was always running.

You always save session state to your appdata folders or settings containers, which are provided by the [Windows.Storage.ApplicationData API](#). Again, we'll see all the details in Chapter 8. What I want to point out here are a few helpers that WinJS provides for all this.

First is the `WinJS.Application.checkpoint` event, which provides a single convenient place to save both session state and any other persistent data you might have.

Second is the `WinJS.Application.sessionState` object. On normal startup, this is just an empty object to which you can add whatever properties you like, including other objects. A typical strategy is to just use this in place of other variables, so there's no need to copy variables into it separately. Within the `checkpoint` event, WinJS automatically serializes the contents of this object (using `JSON.stringify`) into a file within your local appdata folder. Then, when the app is activated with the previous state of `terminated`, WinJS automatically rehydrates the `sessionState` object so that everything you put there is once again available. If you've used this object for storing variables, you only need to avoid settings those values back to their defaults when reloading your state.

Note that because the WinJS ensures that its own handler for `checkpoint` is always called *after* your app gets the event, you can be assured that WinJS will save whatever you write into `sessionState` at any time before your `checkpoint` handler returns.

Third, if you don't want to use the `sessionState` object, the `WinJS.Application` object makes it easy to write your own files without having to use async WinRT APIs. Specifically, it provides (as shown in the [documentation](#)) `local`, `temp`, and `roaming` objects that each have methods called `readText`, `writeText`, `exists`, and `remove`. These objects each work within their respective appdata folders and provide a simplified API for file I/O.

A final aid ties into a deferral mechanism like the one for activation. The deferral is important because Windows will suspend your app as soon as you return from the suspending event, which could be less than five seconds. So, the event args for `WinJS.Application.oncheckpoint` provides a `setPromise` method that ties into the underlying WinRT deferral. As before, you pass a promise for an

async operation (or combined operations) to `setPromise`, which in turn calls the deferral's `complete` method once the promise is fulfilled.

On the WinRT level, the event args for `suspending` contains an instance of `Windows.UI.WebUI.-WebUIApplication.SuspendingOperation`. This provides a `getDeferral` method that returns a deferral object with a `complete` method as with activation.

Well, hey! That sounds pretty good—is this perhaps a sneaky way to circumvent the restriction on running WinRT apps in the background? Will my app keep running indefinitely if I request a deferral by never calling `complete`?

No such luck, amigo. Accept my apologies for giving you a fleeting moment of exhilaration! Deferral or not, five seconds is the *most* you'll ever get. Still, you might want to take full advantage of that time, perhaps to first perform critical async operations (like flushing a cache) and then to attempt other noncritical operations (like a sync to a server) that might greatly improve the user experience. For such purposes, the `suspendingOperation` object also contains a `deadline` property, a `Date` value indicating the time in the future when Windows will forcibly suspend you regardless of any deferral. Once the first operation is complete, you can check if you have time to start start another, and so on.

Basic Session State in Here My Am!

To demonstrate some basic state handling, I've made a few changes to Here My Am! as given in the HereMyAm3c example. Here we have two pieces of information we care about: the variables `lastCapture` (a `StorageFile` with the image) and `lastPosition` (a set of coordinates). We want to make sure we save these when we get suspended so that we can properly apply those values when the app gets launched with the previous state of `terminated`.

With `lastPosition`, we can just move this into the `sessionState` object by prepending `app.sessionState` to the name—in the completed handler for `getGeopositionAsync`, for example:

```
gl.getGeopositionAsync().done(function (position) {
    app.sessionState.lastPosition = {
        latitude: position.coordinate.latitude,
        longitude: position.coordinate.longitude
    };

    updatePosition();
}, function (error) {
    console.log("Unable to get location.");
});
}
```

Because we'll need to set the map location from here and from previously saved coordinates, I've moved that bit of code into a separate function that also makes sure a location exists in `sessionState`:

```
function updatePosition() {
    if (!app.sessionState.lastPosition) {
        return;
    }
}
```

```

    callFrameScript(document.frames["map"], "pinLocation",
    [app.sessionState.lastPosition.latitude, app.sessionState.lastPosition.longitude]);
}

```

Note also that `app.sessionState` is initialized to an empty object by default, `{ }`, so `lastPosition` will be `undefined` until the geolocation call succeeds. This also works to our advantage when rehydrating the app. Here's what the `previousExecutionState` conditions look like for this:

```

if (args.detail.previousExecutionState !==
    activation.ApplicationExecutionState.terminated) {
    //Normal startup: initialize lastPosition through geolocation API
} else {
    //WinJS reloads the sessionState object here. So try to pin the map with the saved location
    updatePosition();
}

```

Because we stored `lastPosition` in `sessionState`, it will have been automatically saved in `WinJS.Application.checkpoint` when the app ran previously. When we restart from `terminated`, WinJS automatically reloads `sessionState`; if we'd saved a value there previously, it'll be there again and `updatePosition` just works.

You can test this by running the app with these changes and then using the *Suspend and shutdown* option on the Visual Studio toolbar. Set a breakpoint on the `updatePosition` call above, and then restart the app in the debugger. You'll see that `sessionState.lastPosition` is initialized at that point.

With the last captured picture, we don't need to save the `StorageFile`, just the pathname: we copied the file into our local appdata (so it persists across sessions already) and can just use the `ms-appdata://` URI scheme to refer to it. When we capture an image, we just save that URI into `sessionState.imageUrl` (the property name is arbitrary) at the end of the promise chain inside `capturePhoto`:

```

that.src = "ms-appdata:///local/HereMyAm/" + newFile.name;
app.sessionState.imageUrl = that.src;

```

This value will also be reloaded when necessary during startup, so we can just initialize the `img src` accordingly:

```

if (app.sessionState.imageUrl) {
    document.getElementById("photo").src = app.sessionState.imageUrl;
}

```

This will initialize the image display from `sessionState`, but we also need to initialize `lastCapture` so that the same image is available through the Share contract. For this we need to also save the full file path so we can re-obtain the `StorageFile` through `Windows.Storage.StorageFile.-getFileFromPathAsync` (which doesn't work with `ms-appdata://` URIs). So, in `capturePhoto`:

```

app.sessionState.imagePath = newFile.path;

```

And during startup:

```

if (app.sessionState.imagePath) {
    Windows.Storage.StorageFile.getFileFromPathAsync(app.sessionState.imagePath)
        .done(function (file) {
            lastCapture = file;

            if (app.sessionState.imageURL) {
                document.getElementById("photo").src = app.sessionState.imageURL;
            }
        });
}

```

I've placed the code to set the `img src` inside the completed handler here because we want the image to appear only if we can also access its `StorageFile` again for sharing. Otherwise the two features of the app would be out of sync.

In all of this, note again that we don't need to explicitly reload these variables within the `terminated` case because WinJS reloads `sessionState` automatically. If we managed our state more directly, such as storing some variables in roaming settings within the `checkpoint` event, we would reload and apply those values at this time.

Data from Services and WinJS.xhr

Though we've seen examples of using data from an app's package (via URLs or `Windows.ApplicationModel.Package.current.installedLocation`) as well as in appdata, it's very likely that your app will incorporate data from a web service and possibly send data to services as well. For this, the most common method is to employ `XmlHttpRequest`. You can use this in its raw (async) form, if you like, or you can save yourself a whole lot of trouble by using the `WinJS.xhr` function, which conveniently wraps the whole business inside a promise.

Making the call is quite easy, as demonstrated in the SimpleXhr example for this chapter. Here we use `WinJS.xhr` to retrieve the RSS feed from the Windows 8 developer blog:

```

WinJS.xhr({ url: "http://blogs.msdn.com/b/windowsappdev/rss.aspx" })
    .done(processPosts, processError, showProgress);

```

That is, give `WinJS.xhr` a URI and it gives back a promise that delivers its results to your completed function (in this case `processPosts`) and will even call a progress function. With the former, the result contains a `responseXML` property, which is a `DomParser` object. With the latter, the event object contains the current XML in its `response` property, which we can easily use to display a download count:

```

function showProgress(e) {
    var bytes = Math.floor(e.response.length / 1024);
    document.getElementById("status").innerText = "Downloaded " + bytes + " KB";
}

```

The rest of the app just chews on the response text looking for `item` elements and extracting and displaying the `title`, `pubDate`, and `link` fields. With a little styling (see `default.css`), and utilizing the

WinJS typography style classes of `win-type-x-large` (for `title`), `win-type-medium` (for `pubDate`), and `win-type-small` (for `link`), we get a quick app that looks like Figure 3-9. You can look at the code to see the details.²¹

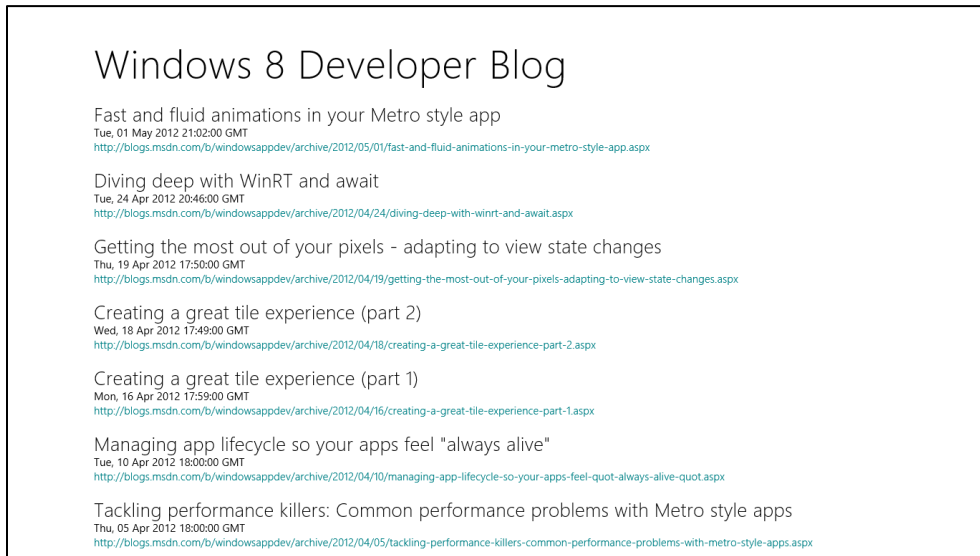


Figure 3-9 The output of the SimpleXhr app.

If you try this app, it's clear that it can use more work, but for a fuller demonstration of XHR and related matters, refer to the [XHR, handling navigation errors, and URL schemes sample](#). You might also be interested in the tutorial called [How to create a mashup](#) in the docs. For the moment, what concerns is not so much the mechanics of talking to services but the implications of suspend and resume.

In particular, an app cannot predict how long it will stay suspended before being resumed or before being terminated and restarted.

In the first case, an app that gets resumed will have all its previous data still in memory. It very much needs to decide, then, whether that data has become stale since the app was suspended or whether sessions with other servers have exceeded their timeout periods. You can also think of it this way: after what period of time will users not remember nor care what was happening the last time they saw your app? If it's a week or longer, it might be reasonable to resume or restart in a default state. Then again, if you pick up right back where they were, users gain increasing confidence that they *can* leave apps

²¹ It's worth mentioning that WinRT has a specific API for dealing with RSS feeds in `Windows.Web.Syndication`. You can use this if you want a more structured means of dealing with such data sources. As it is, JavaScript has intrinsic APIs to work with XML, so it's really your choice. In a case like this, the syndication API along with `Windows.Web.AtomPub` and `Windows.Data.Xml` are very much needed by Windows 8 apps written in other languages that don't have the same built-in features as JavaScript.

running for a long time and not lose anything. Or you can compromise and give the user options to choose from. You'll have to think through your scenario, of course, but if there's any doubt, resume where the app left off.

To check elapsed time, save a timestamp on suspend (from `new Date().getTime()`), get another timestamp in the `resuming` event, take the difference, and compare that against your desired refresh period. A Stock app, for example, might have a very short period. With the Windows 8 developer blog, on the other hand, new posts don't show up more than once a day, so a one-hour period is sufficient to keep up-to-date and to catch new posts within a reasonable timeframe.

This is implemented in SimpleXhr by first placing the `WinJS.xhr` call into a separate function called `downloadPosts`, which is called on startup. Then we register for the resuming event with WinRT:

```
Windows.UI.WebUI.WebUIApplication.onresuming = function () {  
    app.queueEvent({ type: "resuming" });  
}
```

Remember how I said we could use `WinJS.Application.queueEvent` to raise our own events to the app object? Here's a great example. `WinJS.Application` doesn't automatically wrap the `resuming` event because it has nothing to add to that process. But the code below accomplishes exactly the same thing, allowing us to register an event listener right alongside other events like `checkpoint`:

```
app.oncheckpoint = function (args) {  
    //Save in sessionState in case we want to use it with caching  
    app.sessionState.suspendTime = new Date().getTime();  
};  
  
app.addEventListener("resuming", function (args) {  
    //This is a typical shortcut to either get a variable value or a default  
    var suspendTime = app.sessionState.suspendTime || 0;  
  
    //Determine how much time has elapsed in seconds  
    var elapsed = ((new Date().getTime()) - suspendTime) / 1000;  
  
    //Refresh the feed if > 1 hour (or use a small number for testing)  
    if (elapsed > 3600) {  
        downloadPosts();  
    }  
});
```

To test this code, run it in Visual Studio's debugger and set breakpoints within these events. Then click the suspend button in the toolbar (the pause icon shown in Figure 3-7), and you should enter the `checkpoint` handler. Wait a few seconds and click the resume button (play icon), and you should be in the `resuming` handler. You can then step through the code and see that the `elapsed` variable will have the number of seconds that have passed, and if you modify that value (or change 3600 to a smaller number), you can see it call `downloadPosts` again to perform a refresh.

What about launching from the previously terminated state? Well, if you didn't cache any data from before, you'll need to refresh it again anyway. If you do cache some of it, your saved state (such as the

timestamp) helps you decide whether to use the cache or load data anew.

It's worth mentioning here that you can use HTML5 mechanisms like `localStorage`, `indexedDB`, and the app cache for caching purposes; data for these is stored within your local appdata automatically. And speaking of databases, you may be wondering what's available for WinRT apps other than IndexedDB. One option is SQLite, as described in [Using SQLite in a WinRT app](#) (on the blog of Tim Heuer, one of the Windows 8 engineers). You can also use the OData Library for JavaScript that's available from <http://www.odata.org/libraries>. It's one of the easiest ways to communicate with an online SQL Server database (or any other with an OData service), because it just uses XMLHttpRequest under the covers. We'll come back to this topic in Chapter 8.

Handling Network Connectivity (in Brief)

We'll be covering network matters in Chapter 14, "Networking," but there's one important aspect that you should be aware of here. What does an app do with changes to network connectivity, such as disconnection, reconnection, and changes in bandwidth or cost (such as roaming into another provider area)?

The `Windows.Networking.Connectivity` APIs supply the details. There are three main ways to respond to such events:

- First, have a great offline story for when connectivity is lost: cache important data, queue work to be done later, and continue to provide as much functionality as you can without a connection. Clearly this is closely related to your overall state management strategy. For example, if network connectivity was lost while you were suspended, you might not be able to refresh your data at all, so be prepared for that circumstance!
- Second, listen for network changes to know when connectivity is restored, and then process your queues, recache data, and so forth.
- Third, listen for network changes to be cost-aware on metered networks. The [Windows Store certification requirements](#), in fact, have a policy on protecting consumers from "bill shock" caused by excessive data usage on such networks. The last thing you want, to be sure, are negative reviews in the Store on issues like this.

On a simpler note, be sure to test your apps with and without network connectivity to catch little oversights in your code. In *Here My Am!*, for example, my first versions of the script in `map.html` didn't bother to check whether the remote script for Bing Maps had actually been downloaded. Now it checks whether the `Microsoft` namespace (for the `Microsoft.Maps.Map` constructor) is valid. In *SimpleXhr* too, I made sure to provide an error handler to the `WinJS.xhr` promise so that I could at least display a simple message.

Tips and Tricks for WinJS.xhr

Without opening the whole can of worms that is XMLHttpRequest, it's useful here to look at just a

couple of additional points around [WinJS.xhr](#).

First, notice that the single argument to this function is an object that can contain a number of properties. The [url](#) property is the most common, of course, but you can also set the [type](#) (defaults to "GET") and the [responseType](#) for other sorts of transactions, supply [user](#) and [password](#) credentials, set [headers](#) (such as "If-Modified-Since" with a date to control caching), and provide whatever other additional [data](#) is needed for the request (such as query parameters for XHR to a database). You can also supply a [customRequestInitializer](#) function that will be called with the [XmlHttpRequest](#) object just before it's sent, allowing you to perform anything else you need at that moment.

Second is setting a timeout on the request. You can use the [customRequestInitializer](#) for this purpose, setting the [XmlHttpRequest.timeout](#) property and possibly handling the [ontimeout](#) event. Alternately, as we'll see in the "Completing the Promises Story" section at the end of this chapter, you can use the [WinJS.Promise.timeout](#) function, which allows you to set a timeout period after which the [WinJS.xhr](#) promise (and the async operation behind it) will be canceled. Canceling is accomplished by simply calling a promise's [cancel](#) method.

You might have need to wrap [WinJS.xhr](#) in another promise, something that we'll also see at the end of this chapter. You could do this to encapsulate other intermediate processing with the XHR call while the rest of your code just uses the returned promise as usual. In conjunction with a timeout, this can also be used to implement a multiple retry mechanism.

Next, if you need to coordinate multiple XHR calls together, you can use [WinJS.Promise.join](#), which we'll again see later on.

We also saw how to process transferred bytes within the progress handler. You can use other data in the response and request as well. For example, the event args object contains a [readyState](#) property.

For release apps, using XHR with [localhost](#): URI's (local loopback) is blocked by design. During development, however, when this is very useful, for instance, to debug a service without deploying it, you can enable this in Visual Studio by opening the project properties dialog (Project menu -> <project> Properties...), selecting Debugging on the left side, and setting Allow Local Network Loopback to true.

Finally, it's helpful to know that for security reasons cookies are automatically stripped out of XHR responses coming into the local context. One workaround to this is to make XHR calls from a web context [iframe](#) (in which you can use [WinJS.xhr](#)) and then to extract the cookie information you need and pass it to the local context via [postMessage](#). Alternately, you might be able to solve the problem on the service side, such as implementing an API there that will directly provide the information you're trying to extract from the cookies in the first place.

For all other details on this function, refer to the [WinJS.xhr function](#) documentation and its links to associated tutorials.

Page Controls and Navigation

Now we come to an aspect of WinRT apps that very much separates them from typical web applications. In web applications, page-to-page navigation uses `<a href>` hyperlinks or setting `document.location` from JavaScript. This is all well and good; oftentimes there's little or no state to pass between pages, and even when there is, there are well-established mechanisms for doing so, such as HTML5 `sessionStorage` and `localStorage` (which work just fine in WinRT apps).

This type of navigation presents a few problems for WinRT apps, however. For one, navigating to a wholly new page means a wholly new script context—all the JavaScript variables from your previous page will be lost. Sure, you can pass state between those pages, but managing this across an entire app likely hurts performance and can quickly become your least favorite programming activity. It's better and easier, in other words, for client apps to maintain a consistent in-memory state across pages.

Also, the nature of the HTML/CSS rendering engine is such that a blank screen appears when switching pages with a hyperlink. Users of web applications are accustomed to waiting a bit for a browser to acquire a new page (I've found many things to do with 15-second intervals!), but this isn't an appropriate user experience for a fast and fluid WinRT app. Furthermore, such a transition doesn't allow animation of various elements on and off the screen, which can help provide a sense of continuity between pages if that fits with your design.

So, although you can use direct links, WinRT apps typically implement “pages” by dynamically replacing sections of the DOM wholly within the context of a single page like `default.html` (which is akin to how AJAX-based apps work). By doing so, the script context is always preserved and individual elements or groups of elements can be transitioned however you like. In some cases, it even makes sense to simply show and hide pages so that you can switch back and forth quickly. Let's look at the strategies and tools for accomplishing these goals.

WinJS Tools for Pages and Page Navigation

Windows itself, and the app host, provide no mechanism for dealing with pages—from the system's perspective, this is merely an implementation detail for apps to worry about. Fortunately, the engineers who created WinJS and the templates in Visual Studio and Blend worried about this a lot! As a result, they've provided some marvelous tools for managing bits and pieces of HTML+CSS+JS in the context of a single container page:

- [WinJS.UI.Fragments](#) contains a low-level “fragment-loading” API, the use of which is necessary only when you want close control over the process (such as which parts of the HTML fragment get which parent). We won't cover it in this book; see the [documentation](#) and the [Loading HTML Fragments Sample](#).
- [WinJS.UI.Pages](#) is a higher-level API intended for general use and employed by the templates. Think of this as a generic wrapper around the fragment loader that lets you easily define a “page control”—simply an arbitrary unit of HTML, CSS, and JS—that you

can easily pull into the context of another page as you do other controls.²² They are, in fact, implemented like other controls in WinJS (as we'll see in Chapter 4), so you can declare them in markup, instantiate them with `WinJS.UI.processAll`, use as many of them within a single host page as you like, and even nest them.

These APIs provide *only* the means to load and unload individual pages—they pull HTML in from other files (along with referenced CSS and JS) and attach the contents to an element in the DOM. That's it. To actually implement a page-to-page navigation structure, we need two additional pieces: something that manages a navigation stack and something that hooks navigation events to the page-loading mechanism of `WinJS.UI.Pages`.

For the first piece, you can turn to `WinJS.Navigation`, which through about 150 lines of CS101-level code supplies a basic navigation stack. This is all it does. The stack itself is just a list of URIs on top of which `WinJS.Navigation` exposes `state`, `location`, `history`, `canGoBack`, and `canGoForward` properties. The stack is manipulated through the `forward`, `back`, and `navigate` methods, and the `WinJS.Navigation` object raises a few events—`beforenavigate`, `navigating`, and `navigated`—to anyone who wants to listen (through `addEventListener`).²³

For the second piece, you can create your own linkage between `WinJS.Navigation` and `WinJS.UI.Pages` however you like. In fact, in the early stages of app development of Windows 8, even prior to the first public developer preview releases, people ended up writing just about the same boilerplate code over and over. In response, the team at Microsoft responsible for the templates magnanimously decided to supply a standard implementation that also adds some keyboard handling (for forward/back) and some convenience wrappers for layout matters. Hooray!

This piece is called the `PageControlNavigator`. Because it's just a piece of template-supplied code and not part of WinJS, it's entirely under your control, so you can tweak, hack, or lobotomize it however you want.²⁴ In any case, because it's likely that you'll often use the `PageControlNavigator` in your own apps, let's look at how it all works in the context of the Navigation App template.

The Navigation App Template, PageControl Structure, and PageControlNavigator

Taking one step beyond the Blank App template, the Navigation App template demonstrates the basic use of page controls. (The more complex templates build navigation out further.) If you create a new project with this template in Visual Studio or Blend, here's what you'll get:

- **default.html** Contains a single container `div` with a `PageControlNavigator` control

²² If you are at all familiar with user controls in XAML, this is the same idea.

²³ The `beforenavigate` event can be used to cancel the navigation, if necessary. Either call `args.preventDefault` (`args` being the event object), return `true`, or call `args.setPromise` where the promise returns `true`.

²⁴ The [Quickstart: using single-page navigation](#) topic also shows a clever way to hijack HTML hyperlinks and hook them into `WinJS.Navigation.navigate`. This can be a useful tool, especially if you're importing code from a web app.

pointing to “pages/home/home.html”.

- **js/default.js** Contains basic activation and state checkpoint code for the app.
- **css/default.css** Contains global styles.
- **pages/home** Contains a page control for the “home page” contents, composed of **home.html**, **home.js**, and **home.css**. Every page control typically has such markup, script, and style files.
- **js/navigator.js** Contains the implementation of the PageControlNavigator class.

To build upon this structure, add additional pages by using a page control template. I recommend first creating a new folder for the page under *pages*. Then right-click that folder, select Add -> New Item, and select Page Control. This will create suitably named .html, .js, and .css files in that folder.

Now let’s look at the body of default.html (omitting the standard header and a commented-out AppBar control):

```
<body>
  <div id="contenthost" data-win-control="Application.PageControlNavigator"
    data-win-options="{home: '/pages/home/home.html'}"></div>
</body>
```

All we have here is a single container `div` named *contenthost* (it can be whatever you want), in which we declare the `Application.PageControlNavigator` control. With this we specify a single option to identify the first page control it should load (/pages/home/home.html). The `PageControlNavigator` will be instantiated within our `activated` handler’s call to `WinJS.UI.processAll`.

Within home.html we have the basic markup for a page control. This is what the Navigation App template provides as a home page by default, and it’s pretty much what you get whenever you add a new PageControl from the item template:

```
<!DOCTYPE html>
<html>
<head>
  <!--... typical HTML header and WinJS references omitted -->
  <link href="/css/default.css" rel="stylesheet">
  <link href="/pages/home/home.css" rel="stylesheet">
  <script src="/pages/home/home.js"></script>
</head>
<body>
  <!-- The content that will be loaded and displayed. -->
  <div class="fragment homepage">
    <header aria-label="Header content" role="banner">
      <button class="win-backbutton" aria-label="Back" disabled></button>
      <h1 class="titlearea win-type-ellipsis">
        <span class="pagetitle">Welcome to NavApp!</span>
      </h1>
    </header>
```

```

        <section aria-label="Main content" role="main">
            <p>Content goes here.</p>
        </section>
    </div>
</body>
</html>

```

The `div` with *fragment* and *homepage* CSS classes, along with the `header`, creates a page with a standard silhouette and a back button, which the `PageControlNavigator` automatically wires up for keyboard, mouse, and touch events. (Isn't that considerate of it!) All you need to do is customize the text within the `h1` element and the contents within `section`, or just replace the whole smash with the markup you want. (By the way, even though the WinJS files are referenced in each page control, they aren't actually reloaded; they exist here to help you edit a page control in Blend by itself.)

The definition of the actual page control is in `home.js`; by default, the templates just provide the bare minimum:

```

(function () {
    "use strict";

    WinJS.UI.Pages.define("/pages/home/home.html", {
        // This function is called whenever a user navigates to this page. It
        // populates the page elements with the app's data.
        ready: function (element, options) {
            // TODO: Initialize the page here.
        }
    });
})();

```

The most important part is `WinJS.UI.Pages.define`, which associates a relative URI (the page control identifier), with an object containing the page control's methods. Note that the nature of `define` allows you to define different members of the page in multiple places; multiple calls to `WinJS.UI.Pages.define` with the same URI will simply add members to an existing definition (replacing those that already exist).

For a page created with the Page Control item template, you get a couple more methods in the structure (some comments omitted):

```

(function () {
    "use strict";

    WinJS.UI.Pages.define("/page2.html", {
        ready: function (element, options) {
        },

        updateLayout: function (element, viewState, lastViewState) {
            // TODO: Respond to changes in viewState.
        },

        unload: function () {
            // TODO: Respond to navigations away from this page.
        }
    });
})();

```

```
});
})();
```

It's good to note that once you've defined a page control in this way, you can instantiate it from JavaScript with `new` by first obtaining its constructor function from `WinJS.UI.Pages.get(<page_uri>)` and then calling that constructor with the parent element and an object containing its options.

Although a basic structure for the `ready` method is provided by the templates, `WinJS.UI.Pages` and the `PageControlNavigator` will make use of the following if they are available:

PageControl Method	When Called
<code>init</code>	Before elements from the page control have been copied into the DOM.
<code>processed</code>	After <code>WinJS.UI.processAll</code> is complete (that is, controls in the page have been instantiated, which is done automatically), but before page content itself has been added to the DOM.
<code>ready</code>	After the page have been added to the DOM.
<code>error</code>	If an error occurs in loading or rendering the page.
<code>unload</code>	Navigation has left the page.
<code>updateLayout</code>	In response to the <code>window.onresize</code> event, which signals changes between landscape, fill, snap, and portrait view states.

Note that `WinJS.UI.Pages` calls the first four methods; `unload` and `updateLayout`, on the other hand, are used only by the `PageControlNavigator`. Of all of these, the `ready` method is the most common one to implement. It's where you'll do further initialization of control (e.g., populate lists), wire up other page-specific event handlers, and so on. The `updateLayout` method is important when you need to adapt your page layout to new conditions, such as changing the layout of a `ListView` control (as we'll see in Chapter 5, "Collections and Collection Controls").

As for the `PageControlNavigator` itself, the code in `navigator.js` shows how it's defined and how it wires up a few events in its constructor:

```
(function () {
    "use strict";

    // [some bits omitted]
    var nav = WinJS.Navigation;

    WinJS.Namespace.define("Application", {
        PageControlNavigator: WinJS.Class.define(
            // Define the constructor function for the PageControlNavigator.
            function PageControlNavigator (element, options) {
                this.element = element || document.createElement("div");
                this.element.appendChild(this._createPageElement());

                this.home = options.home;
                nav.onnavigated = this._navigated.bind(this);
                window.onresize = this._resized.bind(this);

                document.body.onkeyup = this._keyupHandler.bind(this);
                document.body.onkeypress = this._keypressHandler.bind(this);
                document.body.onmspointerup = this._mspointerupHandler.bind(this);
            }, {
                //...
```

First we see the definition of the `Application` namespace as a container for the `PageControlNavigator` class. Its constructor receives the `element` that contains it (the `contenthost div` in `default.html`) and the `options` declared with `data-win-options` in that element. This control creates another `div` for itself, appends that to its parent, adds a listener for the `WinJS.Navigation.onnavigated` event, and sets up its other listeners. Then it waits for someone to call `WinJS.Navigation.navigate`, which happens in the `activated` handler of `default.js`, to navigate to either the home page or the last page viewed if previous session state was reloaded. When that happens, the `PageControlNavigator`'s `_navigated` handler is invoked, which in turn calls `WinJS.UI.Pages.render` to do the loading, the contents of which are then appended as child elements:

```
_navigated: function (args) {
    var that = this;
    var newElement = that._createPageElement();
    var parentedComplete;
    var parented = new WinJS.Promise(function (c) { parentedComplete = c; });

    args.detail.setPromise(
        WinJS.Promise.timeout().then(function () {
            if (that.pageElement.winControl && that.pageElement.winControl.unload) {
                that.pageElement.winControl.unload();
            }
            return WinJS.UI.Pages.render(args.detail.location, newElement,
                args.detail.state, parented);
        }).then(function parentElement(control) {
            that.element.appendChild(newElement);
            that.element.removeChild(that.pageElement);
            that.navigated();
            parentedComplete();
        })
    );
},
```

One final important point here is that in the page control's JavaScript code, `document` will refer to that page control's contents, not to the content host in `default.html`.

And that, my friends, is how it works! As a concrete example of doing this in a real app, the code in the `HereMyAm3d` sample has been converted to use this model for its single home page. To make this conversion. I started with a new project using the Navigation App template to get the page navigation structures set up. Then I copied or imported the relevant code and resources from `HereMyAm3c`, primarily into the `pages/home/home.html`, `home.js`, and `home.css`. And remember how I said that you could open a page control directly in Blend (which is why pages have WinJS references)? As an exercise, open this project in Blend. You'll first see that everything shows up in `default.html`, but you can also open `home.html` and edit just that page.

You should note that WinJS calls `WinJS.UI.processAll` in the process of loading a page control, so we don't need to concern ourselves with that detail. On the other hand, reloading state when `previousExecutionState==terminated` needs some attention. Because this is picked up in the

`WinJS.Application.onactivated` event *before* any page controls and the `PageControlNavigator` is even instantiated, we need to remember that condition so that the home page's `ready` method can later initialize itself accordingly from `app.sessionState` values. For this we simply write another flag into `app.sessionState` called `initFromState`. We always set this flag on startup, so any value that might be persisted between sessions is irrelevant.

Sidebar: WinJS.Namespace.define and WinJS.Class.define

`WinJS.Namespace.define` provides a shortcut for the JavaScript namespace pattern. This helps to minimize pollution of the global namespace as each app-defined namespace is just a single object in the global namespace but can provide access to any number of other objects, functions, and so on. This is used extensively in WinJS and is recommended for apps as well, where you use a module—that is, `(function() { ... })()`—to define things and then you use a namespace to export selective bits that are referenced through the namespace. In short, use a namespace anytime you're tempted to add any global objects or functions!

The syntax: `var ns = WinJS.Namespace.define(<name>, <members>)` where `<name>` is a string (dots are OK) and `<members>` is any object contained in `{ }`'s. Also, `WinJS.Namespace.defineWithParent(<parent>, <name>, <members>)` defines one within the `<parent>` namespace.

If you call `WinJS.Namespace.define` for the same `<name>` multiple times, the `<members>` are combined. Where collisions are concerned, the most recently added members win. For example:

```
WinJS.Namespace.define("MyNamespace", { x: 10, y: 10 });
WinJS.Namespace.define("MyNamespace", { x: 20, z: 10 });
//MyNamespace == { x: 20, y: 10, z: 10}
```

`WinJS.Class.define` is, for its part, a shortcut for the object pattern, defining a constructor so that objects can be instantiated with `new`.

Syntax: `var className = WinJS.Class.define(<constructor>, <instanceMembers>, <staticMembers>)` where `<constructor>` is a function, `<instanceMembers>` is an object with the class's properties and methods, and `<staticMembers>` is an object with properties and methods that can be directly accessed via `<className>.<member>` (without using `new`).

Variants: `WinJS.Class.derive(<baseClass>, ...)` creates a subclass (`...` is the same arg list as with `define`) using prototypal inheritance, and `WinJS.Class.mix(<constructor>, [<classes>])` defines a class that combines the instance (and static) members of one or more other `<classes>` and initializes the object with `<constructor>`.

Finally, note that because class definitions just generate an object, `WinJS.Class.define` is typically used inside a module with the resulting object exported to the rest of the app through a namespace. Then you can use `new <namespace>.<class>` anywhere in the app.

Sidebar: Helping Out IntelliSense

In WinRT apps you might encounter certain markup structures within code commands, often starting with a triple slash, `///`. These are used by Visual Studio and Blend to provide rich IntelliSense within the code editors. You'll see, for example, `/// <reference path.../>` commands, which create a relationship between your current script file and other scripts, which helps to resolve externally defined functions and variables. This is explained on the [JavaScript IntelliSense](#) page in the documentation. For your own code, especially with namespaces and classes that you will use from other parts of your app, there are comment structures you can use to describe your interfaces to IntelliSense. For details, see [Extending JavaScript IntelliSense](#). If you look around the WinJS JavaScript files themselves, you'll see many examples.

The Navigation Process and Navigation Styles

Having seen how page controls, `WinJS.UI.Pages`, `WinJS.Navigation`, and the `PageControlNavigator` all relate, it's straightforward to see how to navigate between multiple pages within the context of a single HTML page (e.g., `default.html`). With the `PageControlNavigator` instantiated and a page control defined via `WinJS.UI.Pages`, simply call `WinJS.Navigation.navigate` with the relative URI of that page control (its identifier). This loads that page and adds it to the DOM inside the element to which the `PageControlNavigator` is attached. This makes that page visible, thereby “navigating” to it so far as the user is concerned. You can also use the other methods of `WinJS.Navigation` to move forward and back in the nav stack, with its `canGoBack` and `canGoForward` properties allowing you to enable/disable navigation controls. Just remember that all the while, you'll still be in the overall context of your host page where you created the `PageControlNavigator` control.

As an example, create a new project using the Grid App template and look at these particular areas:

- **pages/groupedItems/groupedItems** is the home or “hub” page. It contains a `ListView` control (see Chapter 5) with a bunch of default items.
- Tapping a group header in the list navigates to section page (**pages/groupDetail**). This is done in `groupedItems.html` line 21, where the `click` event calls `WinJS.Navigation.navigate("/pages/groupDetail/groupDetail.html")` with an options argument identifying the specific group to display. That argument comes into the `ready` function of `groupDetail.js`.
- Tapping an item on the hub page goes to detail page (**pages/itemDetail**). The `itemInvoked` handler for the items—see `groupedItems.js` lines 27–37—calls `WinJS.Navigation.navigate("/pages/itemDetail/itemDetail.html")` with an options argument identifying the specific item to display. As with groups, that argument comes into the `ready` function of `itemDetail.js`.
- Tapping an item in the section page also goes to the details page through the same mechanism—see `groupDetail.js` lines 25–28.
- The back buttons on all pages are wired into `WinJS.Navigation.back` by virtue of code in the

PageControlNavigator.

For what it's worth, the Split App template works similarly, where each list item on the items page (pages/items) is wired to navigate to pages/split when invoked.

In any case, the Grid App template also serves as an example of what we call the *Hub-Section-Detail* navigation style. Here the app's home page is the hub where the user can explore the full extent of the app. Tapping a group header navigates to a section, the second level of organization where only items from that group are displayed. Tapping an item (in the hub or in the section) navigates to a details page for that item. You can, of course, implement this navigation style however you like; the Grid App template uses page controls, [WinJS.Navigation](#), and the [PageControlNavigator](#). (Semantic zoom, as we'll see in Chapter 5, is also supported as a navigation tool to switch between hubs and sections.)

An alternate navigation choice is the *Flat* style, which simply has one level of hierarchy. Here, navigation happens to any given page at any time through a *navigation bar* (swiped in from the top edge). When using page controls and [PageControlNavigator](#), navigation controls can just invoke [WinJS.Navigation.navigate](#) for this purpose. Note that in this style, there typically is no back button.

These styles, along with many other UI aspects of navigation, can be found on [Navigation design for WinRT apps](#). This is an essential topic for app designers.

Sidebar: Initial Login and In-App Licensing Agreements (EULA) Pages

Some apps might require either a login or acceptance of a license agreement to do anything, and thus it's appropriate that such pages are the first that appear in an app. In these cases, if the user does not accept a license or doesn't provide a login, the app should display a message describing the necessity of doing so, but it should *always* leave it to the user to close the app if desired. Do not close the app automatically.

Typically, such pages appear only the first time the app is run. If the user provides a valid login, those credentials can be saved for later use via the [Windows.Security.Credentials](#) API. If the user accepts a EULA, that fact should be saved in appdata and reloaded anytime the app needs to check. These settings (login and acceptance of a license) should then always be accessible through the app's Settings charm. Legal notices, by the way, as well as license agreements, should always be accessible through Settings as well. (Note: UX guidance on this should be forthcoming but was not available at the time of writing.)

In both cases, you would typically point to such pages in default.html as the home page. In the [init](#) or [processed](#) methods of the page control, then, which are fired before the page is added to the DOM, check to see if it's not actually necessary to show the page. If that's the case, just call [WinJS.Navigation.navigate](#) to switch over to what will then be the first visible page.

Optimizing Page Switching: Show-and-Hide

Even with page controls, there is still lots going on when navigating from page to page: one set of elements is removed from the DOM, and another is added in. Depending on the pages involved, this can be an expensive operation. For example, if you have a page that displays a list of hundreds or thousands of items, where tapping any item goes to a details page (as with the Grid App template), hitting the back button from a detail page will require reconstruction of the list.

Showing progress indicators can help alleviate the user's anxiety, and the recommendation is to show such indicators after two seconds and provide a means to cancel the operation after ten seconds. Even so, users are notoriously impatient and will likely want to quickly switch between the list and individual items. In this case, page controls might not be the best design.

You could use a split (master-detail) view, of course, but that means splitting the screen real estate. An alternative, then, is to actually keep the list page fully loaded the whole time. Instead of navigating to the item details page in the way we've seen, simply render that details page (see [WinJS.UI.Pages.render](#)) into another `div` that occupies the whole screen and overlays the list, and then make that `div` visible. When you dismiss the details page, just hide the `div` and set `innerHTML` to `""`. This way you get the same effect as navigating between pages but the whole process is much quicker. You can also apply WinJS animations like [enterContent](#) and [exitContent](#) to make the transition more fluid.

The [PageControlNavigator](#) is provided by the templates as part of your app, and you can modify it however you like to provide this kind of capability in a more structured manner.

Completing the Promises Story

Whew! We've taken a long ride in this chapter through many, many fine details of how apps are built and how they run (or don't run!). One consistent theme you may have noticed is that of promises—they've come up in just about every section! Indeed, `async` abounds within both WinJS and WinRT, and thus so do promises.

I wanted to close this chapter, then, by flushing out the story of promises, for they provide richer functionality than we've utilized so far. (If you want the fuller *async* story, read [Keeping apps fast and fluid with asynchrony in the Windows Runtime](#) on the Windows 8 developer blog.)

In review, let's step back for a moment to revisit what a promise really *is*. Simply said, it's an object that returns a value, however complex, sometime in the future. The way you get that value is by calling the promise's `then` or `done` method, whose first parameter is a completed function that will receive the promised value when it is ready—and that function might be called immediately if the result is already available! Furthermore, you can call `then/done` multiple times for the same promise, and you'll just get the same results in each place. This won't cause the system to get confused or anything.

If there's an error along the way, the second parameter to `then/done` is an error function that will be

called instead. (Otherwise exceptions are swallowed by `then` or thrown to the event loop by `done`.)

A third parameter to `then/done` is a `progress` function, which is called periodically by those async operations that support it.²⁵ We've already seen, for instance, how `WinJS.xhr` operations will periodically call the progress function for "ready state" changes and as the response gets downloaded.

Now there's no requirement that a promise has to wrap an async operation or async *anything*. You can, in fact, wrap *any* value in a promise by using the static method `WinJS.Promise.wrap`. Such a wrapper on an already existing value (the future is now!) will just turn right around and call the completed function with that value when you call the promise's `then` or `done` methods. This allows you to use any value, really, where a promise is expected, or return things like errors from functions that otherwise return promises for async operations. (`WinJS.Promise.wraperror` exists for this specific purpose.)

`WinJS.Promise` also provides a host of useful static methods (called directly through `WinJS.Promise`, rather than through a promise object):

- `is` determines whether an arbitrary value is a promise, It makes sure it's an object with a function named "then"; it does not test for "done".
- `as` works like `wrap` except that if you give it a promise, it just returns that promise. If you give a promise to `wrap`, it wraps it in another promise.
- `join` aggregates promises into a single one that's fulfilled when all the values given to it, including other promises, are fulfilled. This essentially groups promises with an AND operation (using `then`, so you'll want to call the join's `done` method to handle errors appropriately).
- `any` is similar to `join` but groups with an OR (again using `then`).
- `cancel` stops an async operation. If an error function is provided, it's called with a value of `Error("canceled")`.
- `theneach` applies completed, error, and progress functions to a group of promises (using `then`), returning the results as another group of values inside a promise.
- `timeout` has a dual nature. If you just give it a timeout value, it returns a promise wrapped around a call to `setTimeout`. If you also provide a promise as the second parameter, it will *cancel* that promise if it's not fulfilled within the timeout period. This latter case is essentially a wrapper for the common pattern of adding a timeout to some other async operation that doesn't have one already.
- `addEventListener/removeEventListener` (and `dispatchEvent`) manage handlers for

²⁵ If you want to impress your friends while reading the documentation, know that if an async function shows it returns a value of type `IAsync[Action | Operation]WithProgress`, then it will utilize a progress function given to a promise. If it only lists `IAsync[Action | Operation]`, progress is not supported.

the `error` event that promises will fire on exceptions (but *not* for cancellation). Listening for this event does not affect use of error functions. It's an addition, not a replacement.²⁶

In addition to using functions like `as` and `wrap`, you can also create a promise from scratch by using `new WinJS.Promise(<init> [, <oncancel>]` where `<init>` is a function that accepts completed, error, and progress callbacks and `oncancel` is an optional function that's called in response to `WinJS.Promise.cancel`.

If `WinJS.Promise.as` doesn't suffice, creating a promise like this is useful to wrap other operations (not just values) within the promise structure so that it can be chained or joined with other promises. For example, if you have a library that talks to a web service through raw `async XMLHttpRequest`, you can wrap each API of that library with promises. You might also use a new promise to combine multiple `async` operations (or other promises!) from different sources into a single promise, where `join` or `any` don't give you the control you need. Another example is encapsulating specific completed, error, and progress functions within a promise, such as to implement a multiple retry mechanism on top of singular XHR operations, to hook into a generic progress updater UI, or to add under-the-covers logging or analytics with service calls so that the rest of your code never needs to know about them.

What We've Just Learned

- How the local and web contexts affect the structure of an app, for pages, page navigation, and `iframe` elements.
- How to use application content URI rules to extend resource access to web content in an `iframe`.
- Using `ms-appdata` URI scheme to reference media content from local, roaming, and temp appdata folders.
- How to execute a series of `async` operations with chained promises.
- How exceptions are handled within chained promises and the differences between `then` and `done`.
- Methods for getting debug output and error reports for an app, within the debugger and the Windows Event Viewer.
- How apps are activated (brought into memory) and the events that occur along the way.
- The structure of app activation code, including activation kinds, previous execution states, and the `WinJS.UI.Application` object.
- Using extended splash screens when an app needs more time to load.
- The important events that occur during an app's lifetime, such as focus events, visibility changes, view state changes, and suspend/resume/terminate.

²⁶ `Async` operations from WinRT that get wrapped in promises do not fire this error event, which is why you typically use an error handler instead.

- The basics of saving and restoring state to restart after being terminated, and the WinJS utilities for implementing this.
- Using data from services through WinJS.xhr and how this relates to the resuming event.
- How to achieve page-to-page navigation within a single page context by using page controls, [WinJS.Navigation](#), and the [PageControlNavigator](#) from the Visual Studio/Blend templates, such as the Navigation App template.
- All the details of promises that are common used with (but not limited to) async operations.

Chapter 4

Controls, Control Styling, and Data Binding

Controls are one of those things you just can't seem to get away from, especially within technology-addicted cultures like those that surround many of us. Even low-tech devices like bicycles and various gardening tools have controls. But this isn't a problem—it's actually a necessity. Controls are the means through which human intent is translated into the realm of mechanics and electronics, and they are entirely made to invite interaction. As I write this, in fact, I'm sitting on an airplane and noticing all the controls that are in my view. The young boy in the row ahead of me seems to be doing the same, and that big "call attendant" button above him is just begging to be pressed!

Controls are certainly essential to Windows 8 apps, and they will invite consumers to poke, prod, touch, click, and swipe them. (They will also invite the oft-soiled hands of many small toddlers as well; has anyone made a dishwasher-safe tablet PC yet?) Windows 8, of course, provides a rich set of controls for apps written in HTML, CSS, and JavaScript. What's most notable in this context is that from the earliest stages of design, Microsoft wanted to avoid forcing HTML/JavaScript developers to use controls that were incongruous with what those developers already know—namely, the use of HTML control elements like `<button>` that can be styled with CSS and wired up in JavaScript by using functions like `addEventListener` and `on<event>` properties.

You can, of course, use those intrinsic HTML controls in a Windows 8 app because those apps run on top of the same HTML/CSS rendering engine as Internet Explorer. No problem. There are even special classes, pseudo-classes, and pseudo-elements that give you fine-grained styling capabilities, as we'll see. But the real question was how to implement Windows 8-specific controls like the toggle switch and list view that would allow you to work with them in the same way—that is, declare them in markup, style them with CSS, and wire them up in JavaScript with `addEventListener` and `on<event>` properties.

The result of all this is that for you, the HTML/JavaScript developer, you'll be looking to WinJS for these controls rather than WinRT. Let me put it another way: if you've noticed the large collection of APIs in the `Windows.UI.Xaml` namespace (which constitutes about 40% of WinRT), guess what? You get to completely ignore all of it! Instead, you'll use the WinJS controls that support declarative markup, styling with CSS, and so on, which means that Windows controls (and custom controls that follow the same model) ultimately show up in the DOM along with everything else, making them accessible in all the ways you already know and understand.

The story of controls in Windows 8 is actually larger than a single chapter. Here we'll be looking primarily at those controls that represent or work with simple data (single values) and that participate

in page layout as elements in the DOM. Participating in the DOM, in fact, is exactly why you can style and manipulate all the controls (HTML and WinJS alike) through standard mechanisms, and a big part of this chapter is to just visually show the styling options you have available. In the latter part of this chapter we'll also explore the related subject of data binding: creating relationships between properties of data objects and properties of controls (including styles) so that the controls reflect what's happening in the data.

The story will then continue in Chapter 5, "Collections and Collection Controls," where we'll look at collection controls—those that work with potentially large data sets—and the additional data-binding features that go with them. We'll also give special attention to media elements (image, audio, and video) in Chapter 10, aptly titled "Media," as they have a variety of unique considerations. Similarly, those elements that are primary for defining layout (like grid and flexbox) are the subject of Chapter 6, "Layout," and we also have a number of UI elements that don't participate in layout at all, like app bars and flyouts, as we'll see in Chapter 7, "Commanding UI."

In short, having covered much of the wiring, framing, and plumbing of an app in Chapter 3, "App Anatomy and Page Navigation," we're ready to start enjoying the finish work like light switches, doorknobs, and faucets—the things that make an app really come to life and engage with human beings.

Sidebar: Essential References for Controls

Before we go on, you'll want to know about two essential topics on the Windows Developer Center that you'll likely refer to time and time again. First is the comprehensive [Controls list](#) that identifies all the controls that are available to you, as we'll summarize later in this chapter. The second are comprehensive [UX Guidelines for Windows 8 apps](#), which describes the best use cases for most controls and scenarios in which not to use them. This is a very helpful resource for both you and your designers.

The Control Model for HTML, CSS, and JavaScript

Again, when Microsoft designed the developer experience for Windows 8, we strove for a high degree of consistency between intrinsic HTML control elements, WinJS controls, and custom controls. I like to refer to all of these as "controls" because they all result in a similar user experience: some kind of widget with which the user interacts with an app. In this sense, every such control has three parts:

- Declarative markup (producing elements in the DOM)
- Applicable CSS (styles as well as special pseudo-class and pseudo-element selectors)
- Methods, properties, and events accessible through JavaScript

Standard HTML controls, of course, already have dedicated markup to declare them, like `<button>`,

`<input>`, and `<progress>`. WinJS and custom controls, lacking the benefit of existing standards, are declared using some root element, typically a `<div>` or ``, with two custom `data-*` attributes: `data-win-control` and `data-win-options`. The value of `data-win-control` specifies the fully qualified name of a public constructor function that creates the actual control as child elements of the root. The second, `data-win-options`, is a JSON string containing key-value pairs separated by commas: `{ <key1>: <value1>, <key1>: <value2>, ... }`.

Hint If you've just made changes to `data-win-options` and your app seems to terminate without reason (and without an exception) when you next launch it, check for syntax errors in the options string. Forgetting the closing `}`, for example, will cause this behavior.

The constructor function itself takes two parameters: the root (parent) element and an options object. Conveniently, `WinJS.Class.define` produce functions that look exactly like this, making it very handy for defining controls (as WinJS does itself). Of course, because `data-*` attributes are, according to the HTML5 specifications, completely ignored by the HTML/CSS rendering engine, some additional processing is necessary to turn an element with these attributes into an actual control in the DOM. And this, as I've hinted at before, is exactly the life purpose of the `WinJS.UI.process` and `WinJS.UI.processAll` methods. As we'll see shortly, these methods parse the options attribute and pass the resulting object and the root element to the constructor function identified in `data-win-control`.

The result of this simple declarative markup plus `WinJS.UI.process/processAll` is that WinJS and custom controls are just elements in the DOM like any others. They can be referenced by DOM-traversal APIs and targeted for styling using the full extent of CSS selectors (as we'll see in the styling gallery later on). They can listen for external events like other elements and can surface events of their own by implementing `[add/remove]EventListener` and `on<event>` properties. (WinJS again provides standard implementations of `addEventListener`, `removeEventListener`, and `dispatchEvent` for this purpose.)

Let's now look at the controls we have available for Windows 8 apps, starting with the HTML controls and then the WinJS controls. In both cases we'll look at their basic appearance, how they're instantiated, and the options you can apply to them.

HTML Controls

HTML controls, I hope, don't need much explaining. They are described in HTML5 references, such as http://www.w3schools.com/html5/html5_reference.asp, and shown with default "light" styling in Figure 4-1 and Figure 4-2. (See the next section for more on WinJS stylesheets.) It's worth mentioning that most embedded objects are not supported, except for a specific ActiveX controls; see [Migrating a web app](#).

Creating or instantiating an HTML also works as you would expect. You either declare them in

markup (using attributes to specify options, the rundown of which is given in the table following Figure 4-2), or you create them procedurally from JavaScript by calling [new](#) with the appropriate constructor, configuring properties and listeners as desired, and adding the element to the DOM wherever its needed. Nothing new here at all where Windows 8 apps are concerned.

For examples of creating and using these controls, refer to the [Common HTML Controls sample](#) in the Windows SDK, from which the images in Figure 4-1 and Figure 4-2 were obtained.

Button (<button> <input type="button">)



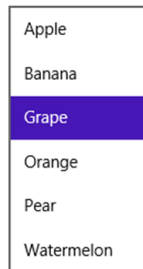
Checkbox (<input type="checkbox">)



Drop-Down
(<select>)



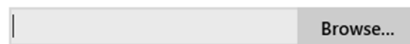
ListBox
(<select>)



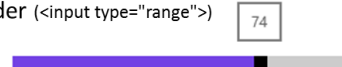
Hyperlink (<a href>, <link>)

<http://dev.windows.com>

File Upload (<input type="file">)



Slider (<input type="range">)



Progress Bar (<progress>)



Progress Ring (<progress>)



Radio Button (<input type="radiobutton">)



Figure 4-1 Standard HTML5 controls with default "light" styles (the ui-light.css stylesheet of WinJS).

Single-line text (<input>;
all HTML5 types are supported)

Single line text input
<input type="text" />

Clear Button shows when entering text;
oninput event fires when pressed.

Password input
<input type="password" />

Reveal button (shows password characters)

Number input
<input type="number" />

Email input
<input type="email" />

Phone number input
<input type="tel" />

URL input
<input type="url" />

Multi-line text (<textarea>)

Multi-line text input
<textarea></textarea>

Rich text (<div>)

Multi-line rich text input
<div> with contentEditable="true" and some
custome styles.

Select some text and then click the "Bold" button.

Figure 4-2 Standard HTML5 text input controls with default "light" styles (the ui-light.css stylesheet of WinJS).

Control	Markup	Common Option Attributes	Element Content (inner text/HTML)
Button	<button type="button">	(note that without type, the default is "submit")	Button text
Button	<input type="button"> <input type="submit"> <input type="reset">	value (button text)	n/a
Checkbox	<input type="checkbox">	value, checked	n/a (use a label element around the input control to add clickable text)
Drop Down List	<select>	size="1" (default), multiple, selectedIndex	Multiple <option> elements
Email	<input type="email">	value (initial text)	n/a

File Upload	<code><input type="file"></code>	accept (mime types), multiple	n/a
Hyperlink	<code><a></code>	href, target	Link text
ListBox	<code><select></code> with size > 1	size (a number greater than 1), multiple, selectedIndex	Multiple <code><option></code> elements
Multi-line Text	<code><textarea></code>	cols, rows, readonly	Initial text content
Number	<code><input type="number"></code>	value (initial text)	n/a
Password	<code><input type="password"></code>	value (initial text)	n/a
Phone Number	<code><input type="tel"></code>	value (initial text)	n/a
Progress	<code><progress></code>	value (initial position), max (highest position; min is 0); no value makes it indeterminate	n/a
Radiobutton	<code><input type="radio"></code>	value, checked, defaultChecked	Radiobutton label
Rich Text	<code><div></code>	contentEditable="true"	HTML content
Slider	<code><input type="range"></code>	min, max, value (initial position), step (increment)	n/a
URI	<code><input type="url"></code>	value (initial text)	n/a

Two areas that add something to HTML controls are the WinJS stylesheets and the additional methods, properties, and events that Microsoft's rendering engine adds to most HTML elements. These are the subjects of the next two sections.

WinJS stylesheets: ui-light.css, ui-dark.css, and win-* styles

WinJS comes with two parallel stylesheets that provide many default styles and style classes for WinRT apps: ui-light.css and ui-dark.css. You'll always use one or the other, as they are mutually exclusive. The first is intended for apps that are oriented around text, because dark text on a light background is generally easier to read (so this theme is often used for news readers, books, magazines, etc., including figures in published books like this!). The dark theme, on the other hand, is intended for media-centric apps like picture and video viewers where you want the richness of the media to stand out.

Both stylesheets define a number of `win-*` style classes, which I like to think of as style packages that effectively add styles and CSS-based behaviors (like the `:hover` pseudo-class) that turn standard HTML controls into a Windows 8-specific variant. These are `win-backbutton` for buttons, `win-ring`, `win-medium`, and `win-large` for circular `progress` controls, `win-small` for a rating control, `win-vertical` for a vertical slider (range) control, and `win-textarea` for a content editable `div`. If you want to see the details, search on their names in the Style Rules tab in Blend.

Extensions to HTML Elements

As you probably know already, there are many developing standards for HTML and CSS. Until these are brought to completion, implementations of those standards in various browsers are typically made available ahead of time with vendor-prefixed names. In addition, browser vendors sometimes add their own extensions to the DOM API for various elements.

With WinRT apps, of course, you don't need to worry about the variances between browsers, but since these apps essentially run on top of the Internet Explorer engines, it helps to know about those extensions that still apply. These are summarized in the table below, and you can find the full [Elements Reference](#) in the documentation for all the details your heart desires (and too much to spell out here).

If you've been working with HTML5 and CSS3 in Internet Explorer already, you might be wondering why the table doesn't show the various animation (`msAnimation*`), transition (`msTransition*`), and transform properties (`msPerspective*` and `msTransformStyle`), along with `msBackfaceVisibility`. This is because these standards are now far enough along that they no longer need vendor prefixes with Internet Explorer 10 or WinRT apps (though the `ms*` variants still work).

Methods	Description
<code>msMatchesSelector</code>	Determines if the control matches a selector.
<code>ms[Set Get Release]PointerCapture</code>	Captures, retrieves, and releases pointer capture for an element.
Style properties (on <code>element.style</code>)	Description
<code>msGrid*</code> , <code>msRow*</code>	Gets or sets placement of element within a CSS grid.
Events (add "on" for event properties)	Description
<code>mscontentzoom</code>	Fires when a user zooms an element (Ctrl+ +/-, Ctrl + mousewheel), pinch gestures.
<code>msgesture[change end hold tap pointercapture]</code>	Gesture input events (see Chapter 9, "Input and Sensors").
<code>msinertiastart</code>	Gesture input events (see Chapter 9).
<code>mslostpointercapture</code>	Element lost capture (set previously with <code>msSetPointerCapture</code>).
<code>mspointer[cancel down hover move out over up]</code>	Pointer input events (see Chapter 9).
<code>msmanipulationstatechanged</code>	State of a manipulated element has changed.

WinJS Controls

Windows 8 defines a number of controls that help apps fulfill WinRT app design guidelines. As noted before, these are implemented in WinJS for WinRT apps written in HTML, CSS, and JavaScript, rather than WinRT; this allows those controls to integrate naturally with other DOM elements. Each control is

defined as part of the `WinJS.UI` namespace using `WinJS.Class.define`, where the constructor name matches the control name. So the full constructor name for a control like the Rating is `WinJS.UI.Rating`.

The simpler controls that we'll cover here in this chapter are `DatePicker`, `Rating`, `ToggleSwitch`, and `Tooltip`, the default styling for which are shown in Figure 4-3. The collection controls that we'll cover in Chapter 5 are `FlipView`, `ListView`, and `SemanticZoom`. App bars, flyouts, and others that don't participate in layout are again covered in later chapters. Apart from these, there is only one other, `HtmlControl`, which is simply an older (and essentially deprecated) alias for `WinJS.UI.Pages`. That is, the `HtmlControl` is the same thing as rendering a page control: it's an arbitrary block of HTML, CSS, and JavaScript that you can declaratively incorporate anywhere in a page. We've already discussed all those details in Chapter 3, so there's nothing more to add here.

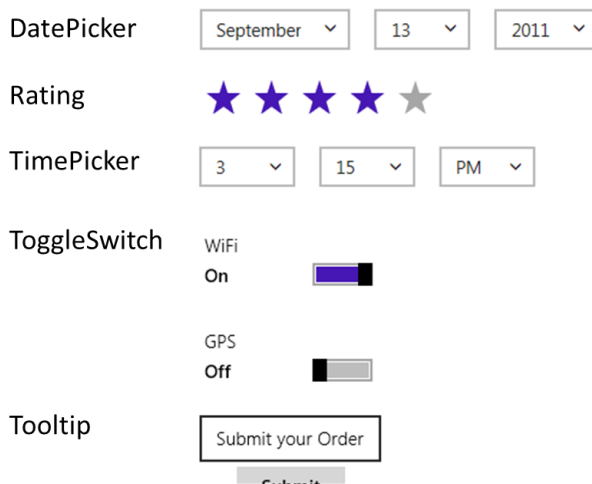


Figure 4-3 Default (light) styles on the simple WinJS controls.

The `WinJS.UI.Tooltip` control, you should know, can utilize any HTML including other controls, so it goes well beyond the plain text tooltip that HTML provides automatically for the `title` attribute. We'll see more examples later.

So again, a WinJS control is declared in markup by attaching `data-win-control` and `data-win-options` attributes to some root element. That element is typically a `div` (block element) or `span` (inline element), because these don't bring much other baggage, but any element can be used. These elements can, of course, have `id` and `class` attributes as needed. The available options for these controls are summarized in the table below, which includes those events that can be wired up through the `data-win-options` string, if desired. For full documentation on all these options, start with the [Controls list](#) in the documentation and go to the control-specific topics linked from there.

Fully-qualified constructor name in data-win-control	Options in data-win-options
WinJS.UI.DatePicker	Properties: <code>calendar</code> , <code>current</code> , <code>datePattern</code> , <code>disabled</code> , <code>maxYear</code> , <code>minYear</code> , <code>monthPattern</code> , <code>yearPattern</code> Events: <code>onchange</code>
WinJS.UI.Rating	Properties: <code>averageRating</code> , <code>disabled</code> , <code>enableClear</code> , <code>maxRating</code> , <code>tooltipStrings</code> (an array of strings the size of <code>maxRating</code>), <code>userRating</code> Events: <code>oncancel</code> , <code>onchange</code> , <code>onpreviewchange</code>
WinJS.UI.TimePicker	Properties: <code>clock</code> , <code>current</code> , <code>disabled</code> , <code>hourPattern</code> , <code>minuteIncrement</code> , <code>periodPattern</code> Events: <code>onchange</code>
WinJS.UI.ToggleSwitch	Properties: <code>checked</code> , <code>disabled</code> , <code>labelOff</code> , <code>labelOn</code> , <code>title</code> Events: <code>onchange</code>
WinJS.UI.Tooltip	Properties: <code>contentElement</code> , <code>innerHTML</code> , <code>infoTip</code> , <code>extraClass</code> , <code>placement</code> Events: <code>onbeforeclose</code> , <code>onbeforeopen</code> , <code>onclosed</code> , <code>onopened</code> Methods: <code>open</code> , <code>close</code>

Again, the `data-win-options` string containing key-value pairs, one for each property or event, separated by commas, in the form `{ <key1>: <value1>, <key1>: <value2>, ... }`. For events, whose names in the options string always start with `on`, the value is the name of the event handler you want to assign.

In JavaScript code, you can also assign event handlers by using `<element>.addEventListener("event", ...)` where `<element>` is the element for which the control was declared and `<event>` drops the “on” as usual. To access the properties and events directly, use `<element>.winControl.<property>`. The `winControl` object is created when the WinJS control is instantiated and attached to the element, so that’s where these options are available.

WinJS Control Instantiation

As we’ve seen a number of times already, WinJS controls declared in markup with `data-*` attributes are not instantiated until you call `WinJS.UI.process(<element>)` for a single control or `WinJS.UI.processAll` for all such elements in the DOM. To understand this process, here’s what `WinJS.UI.process` does for a single element:

13. Parse the `data-win-options` string into an options object.
14. Extract the constructor specified in `data-win-control`, and then call `new` on that function passing the root element and the options object.
15. The constructor creates whatever child elements it needs within the root element.
16. The object returned from the constructor—the control object—is stored in the root element’s `winControl` property.

Clearly, then, the bulk of the work really happens in the constructor. Once this has happened, other JavaScript code (as in your activated method) can call methods, manipulate properties, and add

listeners for events on both the root element and the `winControl` object. The latter, clearly, must be used for WinJS control-specific methods, properties, and events.

`WinJS.UI.processAll`, for its part, simply traverses the DOM looking for data-win-control attributes and does `WinJS.UI.process` for each. How you use both of these is really your choice: `processAll` goes through a whole page (or just a page control—whatever the document object refers to), whereas `process` lets you control the exact sequence or instantiate controls for which you dynamically insert markup. Note that in both cases the return value is a promise, so if you need to take additional steps after processing is complete, call the promise's `done` method with a suitable completed function.

It's also good to understand that `process` and `processAll` are really just helper functions. If you need to, you can just directly call `new` on a control constructor with an element and options object. This will create the control and attach it to the given element automatically. You can also pass `null` for the element, in which case the WinJS control constructors create a new `div` element to contain the control that is otherwise unattached to the DOM. This would allow you, for instance, to build up a control offscreen and attach it to the DOM only when needed.

To see all this in action, we'll look at some examples with both the Rating and Tooltip controls in a moment. First, however, we need to discuss a matter referred to as *strict processing*.

Strict Processing and processAll Functions

WinJS has three DOM-traversing functions: `WinJS.UI.processAll`, `WinJS.Binding.processAll` (which we'll see later in this chapter), and `WinJS.Resources.processAll` (which we'll see in Chapter 17, "Apps for Everyone"). Each of these looks for specific `data-win-*` attributes and then takes additional actions using those contents. Those actions, however, can involve calling a number of different types of functions:

- Functions appearing in a "dot path" for control processing and binding sources
- Functions appearing in the left-hand side for binding targets, resource targets, or control processing
- Control constructors and event handlers
- Binding initializers or functions used in a binding expression
- Any custom layout used for a ListView control

Such actions introduce a risk of injection attack if a `processAll` function is called on untrusted HTML, such as arbitrary markup obtained from the web. To mitigate this risk, WinJS has a notion of strict processing that is enforced within all HTML/JavaScript apps.. The effect of strict processing is that any functions indicated in markup that `processAll` methods might encounter must be "marked for processing" or else processing will fail. The mark itself is simply a `supportedForProcessing` property on the function object that is set to `true`.

Functions returned from `WinJS.Class.define`, `WinJS.Class.derive`, `WinJS.UI.Pages.define`, and `WinJS.Binding.converter` are automatically marked in this manner. For other functions, you can either set a `supportedForProcessing` property to true directly or use marking functions like so:

```
WinJS.Utilities.markSupportedForProcessing(myfunction);
WinJS.UI.eventHandler(myHandler);
WinJS.Binding.initializer(myInitializer);

//Also OK
<namespace>.myfunction = WinJS.UI.eventHandler(function () {
});
```

Note also that appropriate functions coming directly from WinJS, such as all `WinJS.UI.*` control constructors, as well as `WinJS.Binding.*` functions, are marked by default.

So, if you reference custom functions from your markup, be sure to mark them accordingly. But this is *only* for references from *markup*: you don't need to mark functions that you assign to `on<event>` properties in code or pass to `addEventListener`.

Example: WinJS.UI.Rating Control

OK, now that we got the strict processing stuff covered, let's see some concrete example of working with a WinJS control.

For starters, here's some markup for a `WinJS.UI.Rating` control, where the options specify two initial property values and an event handler:

```
<div id="rating1" data-win-control="WinJS.UI.Rating"
    data-win-options="{averageRating: 3.4, userRating: 4, onchange: changeRating}">
</div>
```

To instantiate this control, we need to call either of the following functions:

```
WinJS.UI.process(document.getElementById("rating1"));
WinJS.UI.processAll();
```

Again, both of these functions return a promise, but it's unnecessary to call `done` unless we need to do additional post-instantiation processing or surface exceptions that might have occurred (and that are otherwise swallowed). Also, note that the `changeRating` function specified in the markup must be globally visible and marked for processing, or else the control will fail to instantiate.

Next, we can instantiate the control and set the options procedurally. So, in markup:

```
<div id="rating1" data-win-control="WinJS.UI.Rating"></div>
```

And in code:

```
var element = document.getElementById("rating1");
WinJS.UI.process(element);
element.winControl.averageRating = 3.4;
element.winControl.userRating = 4;
```

```
element.winControl.onChange = changeRating;
```

The last three lines above could also be written as follows using the `WinJS.UI.setOptions` method, but this isn't recommended because it's harder to debug:

```
var options = { averageRating: 3.4, userRating: 4, onChange: changeRating };
WinJS.UI.setOptions(element.winControl, options);
```

We can also just instantiate the control directly. In this case the markup is nonspecific:

```
<div id="rating1"></div>
```

And we call `new` on the constructor ourselves:

```
var newControl = new WinJS.UI.Rating(document.getElementById("rating1"));
newControl.averageRating = 3.4;
newControl.userRating = 4;
newControl.onChange = changeRating;
```

Or, as mentioned before, we can skip the markup entirely, have the constructor create an element for us (a `div`), and attach it to the DOM at our leisure:

```
var newControl = new WinJS.UI.Rating(null,
    { averageRating: 3.4, userRating: 4, onChange: changeRating });
newControl.element.id = "rating1";
document.body.appendChild(newControl.element);
```

Hint If you see strange errors on instantiation with these latter two cases, check whether you forgot the `new` and are thus trying to invoke the constructor function directly.

Note also in these last two cases that the `rating1` element will have a `winControl` property that is the same as `newControl` as returned from the constructor.

To see this control in action, please refer to the [HTML Rating control sample](#) in the SDK.

Example: WinJS.UI.Tooltip Control

With most of the other simple controls—namely the `DatePicker`, `TimePicker`, and `ToggleSwitch`—you can work with them in the same ways as we just saw with Ratings. All that changes are the specifics of their properties and events; again, start with the [Controls list](#) page and navigate to any given control for all the specific details. Also, for working samples refer to the [HTML DatePicker and TimePicker controls](#) and the [HTML ToggleSwitch control](#) samples in the SDK.

The `WinJS.UI.Tooltip` control is a little different, however, so I'll illustrate its specific usage. First, to attach a tooltip to a specific element, you can either add a `data-win-control` attribute to that element or place the element itself inside the control:

```
<!-- Directly attach the Tooltip to its target element -->
<targetElement data-win-control="WinJS.UI.Tooltip">
</targetElement>
```

```

<!-- Place the element inside the Tooltip -->
<span data-win-control="WinJS.UI.Tooltip">
    <!-- The element that gets the tooltip goes here -->
</span>

<div data-win-control="WinJS.UI.Tooltip">
    <!-- The element that gets the tooltip goes here -->
</div>

```

Second, the `contentElement` property of the tooltip control can name another element altogether, which will be displayed when the tooltip is invoked. For example, if we have this piece of hidden HTML in our markup (and notice that it contains other controls):

```

<div style="display: none;">
    <!--Here is the content element. It's put inside a hidden container
    so that it's invisible to the user until the tooltip takes it out.-->
    <div id="myContentElement">
        <div id="myContentElement_rating">
            <div data-win-control="WinJS.UI.Rating" class="win-small movieRating"
                data-win-options="{userRating: 3}">
            </div>
        </div>
        <div id="myContentElement_description">
            <p>You could provide any DOM element as content, even with WinJS controls inside. The tooltip
            control will re-parent the element to the tooltip container, and block interaction events on that element,
            since that's not the suggested interaction model.</p>
        </div>
        <div id="myContentElement_picture">
        </div>
    </div>
</div>

```

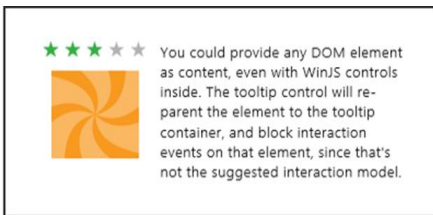
we can reference it like so:

```

<div data-win-control="WinJS.UI.Tooltip"
    data-win-options="{infotip: true, contentElement: myContentElement}">
    <span>My piece of data</span>
</div>

```

When you hover over the text (with a mouse or hover-enabled touch hardware), this tooltip will appear:



This example is taken directly from the [HTML Tooltip control sample](#) in the SDK, so you can go there to see how all this works directly.

Working with Controls in Blend

Before we move onto the subject of control styling, it's a good time to highlight a few additional features of Blend for Visual Studio where controls are concerned. As I mentioned in Video 2-1, the Assets tab in Blend gives you quick access to all the HTML elements and WinJS controls (among many other elements) that you can just drag and drop into whatever page is showing in the artboard. (See Figure 4-4.) This will create basic markup, such as a `div` with a `data-win-control` attribute for WinJS controls; then you can go to the HTML Attributes pane (on the right) to set options in the markup. (See Figure 4-5.)

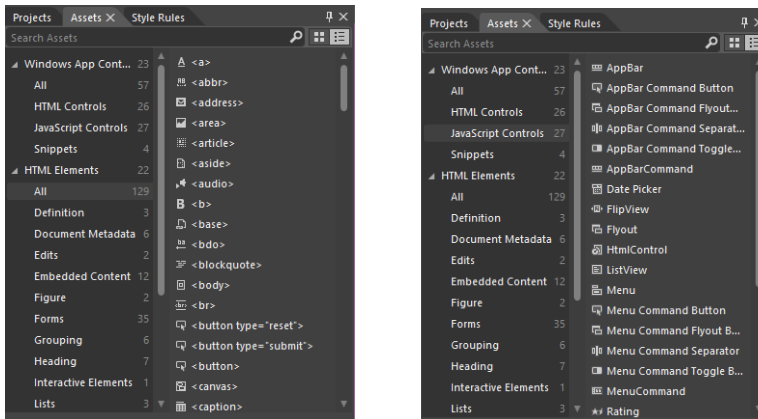


Figure 4-4 HTML elements (left) and WinJS control (right) as shown in Blend's Assets tab.

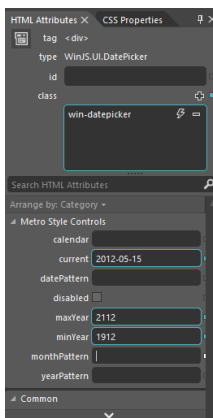


Figure 4-5 Blend's HTML Attributes tab shows WinJS control options, and editing them will affect the `data-win-options` attribute in markup.

Next, take a moment to load up the [Common HTML Controls sample](#) from the SDK into Blend. This is a great opportunity to try out Blend's Interactive Mode to navigate to a particular page and explore

the interaction between the artboard and the Live DOM. (See Figure 4-6.) Once you open the project, go into interactive mode by selecting View -> Interactive Mode on the menu, pressing Ctrl+Shift+I, or clicking the small leftmost button on the upper right corner of the artboard. Then select scenario 5 (Progress introduction) in the listbox, which will take you to the page shown in Figure 4-6. Then exit interactive mode (same commands), and you'll be able to click around on that page. A short demonstration of using interactive mode in this way is given in Video 4-1 in this chapter's companion content.

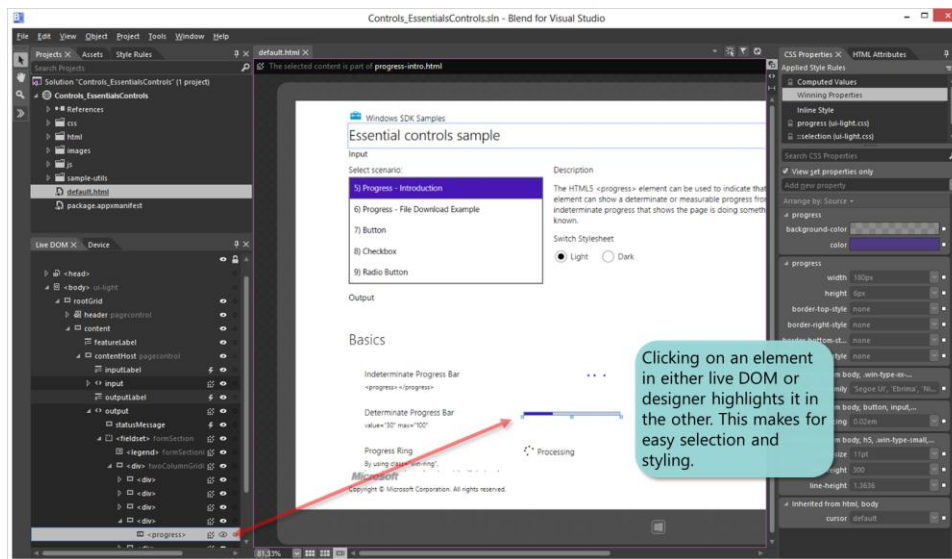


Figure 4-6 Blend's interaction between the artboard and the Live DOM.

With the Common HTML Controls sample, you'll see that there's just a single element in the Live DOM for intrinsic controls, as there should be, since all the internal details are part and parcel of the HTML/CSS rendering engine. On the other hand, load up the [HTML Rating control sample](#) instead and expand the div that contains one such control. There you'll see all the additional child elements that make up this control (shown in Figure 4-7), and you can refer to the right-hand pane for HTML attributes and CSS properties. You can see something similar (with even more detailed information), in the DOM Explorer of Visual Studio when the app is running. (See Figure 4-8.)

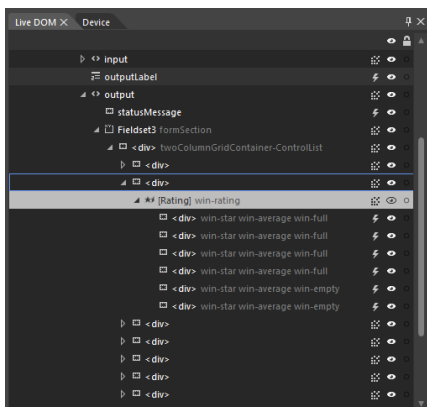


Figure 4-7 Expanding a WinJS control in Blend’s Live DOM reveals the elements that are used to build it.

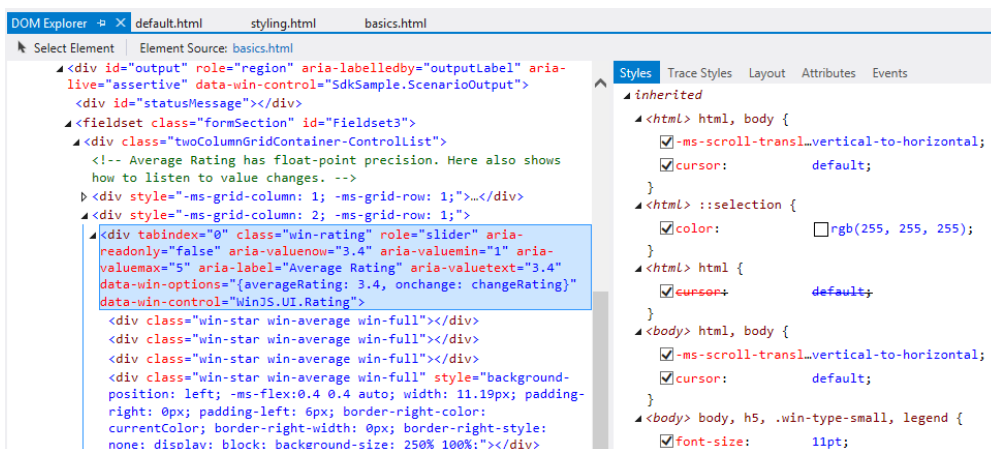


Figure 4-8 Expanding a WinJS control in Visual Studio’s DOM Explorer also shows complete details for a control.

Control Styling

Now we come to a topic where we’ll mostly get to look at lots of pretty pictures: the various ways in which HTML and WinJS controls can be styled. As we’ve discussed, this happens through CSS all the way, either in a stylesheet or by assigning `style.*` properties, meaning that apps have full control over the appearance of controls. In fact, absolutely *everything* that’s different between HTML controls in a WinRT app and the same controls on a web page, is due to styling and styling alone.

For both HTML and WinJS controls, CSS standards apply including pseudo-selectors like `:hover`, `:active`, `:checked`, and so forth, along with `-ms-*` prefixed styles for emerging standards.

For HTML controls, there are also additional `-ms-*` styles—that aren’t part of CSS3—to isolate specific parts of those controls. That is, because the constituent parts of such controls don’t exist

separately in the DOM, pseudo-selectors—like `::-ms-check` to isolate a checkbox mark and `::-ms-fill-lower` to isolate the left or bottom part of a slider—allow you to communicate styling to the depths of the rendering engine. In contrast, all such parts of WinJS controls are addressable in the DOM, so they are just styled with specific `win-*` classes defined in the WinJS stylesheets. That is, the controls are simply rendered with those style classes. Default styles are defined in the WinJS stylesheets, but apps can override any aspect of those to style the controls however you want.

In a few cases, as already pointed out, certain `win-*` classes define style packages for use with HTML controls, such as `win-backbutton`, `win-vertical` (for a slider) and `win-ring` (for a progress control). These are intended to style standard controls to look like special system controls.

There are also a few general purpose `-ms-*` styles (not selectors) that can be applied to many controls (and elements in general), along with some general WinJS `win-*` style classes. These are summarized in Table 4-4.

Style or Class	Description
<code>-ms-user-select: none inherit element text auto</code>	Enables or disables selection for an element. Setting to 'none' is particularly useful to prevent selection in text elements.
<code>-ms-zoom: <percentage></code>	Optical zoom (magnification).
<code>-ms-touch-action: auto none</code> (and more)	Allows specific tailoring of a control's touch experience, enabling more advanced interaction models.
<code>win-interactive</code>	Prevents default behaviors for controls contained inside FlipView and ListView controls (see Chapter 5).
<code>win-swipeable</code>	Sets <code>-ms-touch-action</code> styles so a control within a ListView can be swiped (to select) in one direction without causing panning in the other.
<code>win-small</code> , <code>win-medium</code> , <code>win-large</code>	Size variations to some controls.
<code>win-textarea</code>	Sets typical text editing styles.

Table 4-4 General `-ms-*` styles and `win-*` classes for WinRT apps.

For all of these and more, spend some time with these three reference topics: [WinJS styles for typography](#), [WinJS CSS classes for HTML controls](#), and [WinJS classes for WinJS controls](#). I also wanted to provide you with a summary (Table 4-5) of all the other vendor-prefixed styles (or selectors) that are supported within the CSS engine for WinRT apps. I made this list because the documentation here can be hard to penetrate: you have to click through the individual pages under the [Cascading Style Sheets](#) topic in the docs to see what little bits have been added to the CSS you already know.

Area	Styles
Backgrounds and borders	<code>-ms-background-position-[x y]</code>
Box model	<code>-ms-overflow-[x y]</code>
Basic UI	<code>-ms-text-overflow</code> (for ellipses rendering) <code>-ms-user-select</code> (sets or retrieves where users are able to select text within an element.) <code>-ms-zoom</code> (optical zoom)
Flexbox	<code>-ms-[inline-]flexbox</code> (values for <code>display</code>); <code>-ms-flex</code> and <code>-ms-flex-[align direction order pack wrap]</code>

Gradients	<code>-ms-[repeating-]linear-gradient</code> , <code>-ms-[repeating-]radial-gradient</code>
Grid	<code>-ms-grid</code> and <code>-ms-grid-[column column-align columns column-span grid-layer row row-align rows row-span]</code>
High contrast	<code>-ms-high-contrast-adjust</code>
Regions	<code>-ms-flow-[from into]</code> along with the <code>MSRangeCollection</code> method
Text	<code>-ms-block-progression</code> , <code>-ms-hyphens</code> and <code>-ms-hyphenate-limit-[chars lines zone]</code> , <code>-ms-text-align-last</code> , <code>-ms-word-break</code> , <code>-ms-word-wrap</code> , <code>-ms-ime-mode</code> , <code>-ms-layout-grid</code> and <code>-ms-layout-grid-[char line mode type]</code> , and <code>-ms-text-[autospace kashida-space overflow underline-position]</code>
Other	<code>-ms-writing-mode</code>

Table 4-5 Summary of `-ms-*` styles beyond CSS standards. Vendor-prefixed styles for animations, transforms, and transitions are still supported, though no longer necessary, because these standards have recently been finalized.

Styling Gallery: HTML Controls

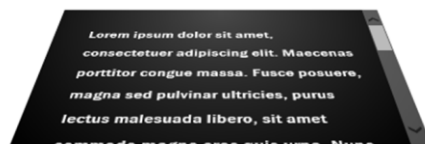
Now we get to enjoy a visual tour of styling capabilities for WinRT apps. Much can be done with standard styles, and then there are all the things you can do with special styles and classes as shown in the graphics in this section. The specifics of all these examples can be seen in the [Common HTML Controls sample](#) in the SDK.

Also check out the very cool [Applying app theme color \(theme roller\) sample](#). This beauty lets you configure the primary and secondary colors for an app, shows how those colors affect different controls, and produces about 200 lines of precise CSS that you can copy into your own stylesheet. This very much helps you create a color theme for your app, which we very much encourage to establish an app's own personality within the overall Windows 8 design guidelines.

Button (background-color)



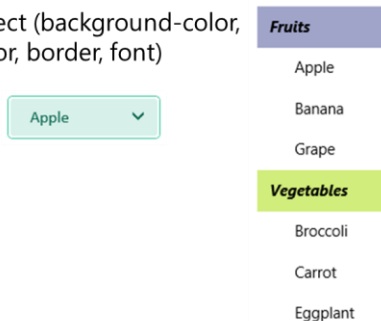
Text Area (transform)



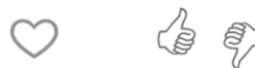
Progress (color)



Select (background-color, color, border, font)



Checkbox/Radiobutton (background-image and :checked)



Button

```
<button class="win-backbutton">
```

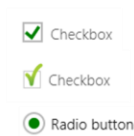


Checkbox/Radiobutton

```
CSS pseudo-element: input[type="checkbox"].<class>::-ms-check (color)
```

```
CSS pseudo-element: input[type="checkbox"].<class>::-ms-check (image)
```

```
CSS pseudo-element: input[type="radio"].<class>::-ms-check (color)
```



File upload

```
CSS pseudo-element: input[type="file"].<class>::-ms-value
```



```
CSS pseudo-element: input[type="file"].<class>::-ms-browse
```

Text Input (most forms)

```
CSS pseudo-element: input[type="text"].<class>::-ms-value
```

```
CSS pseudo-class: input[type="text"].<class>::-ms-input-placeholder
```



CSS background image (and other styles)

```
CSS pseudo-element: input[type="text"].<class>::-ms-clear
```

Progress

```
CSS pseudo-element:
progress.<class>::-ms-fill {
  -ms-animation-name: -ms-ring;
}
```



```
CSS pseudo-element: progress.<class>::-ms-fill (background-image, etc)
```



Text Input (password)

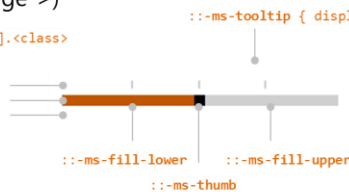
```
CSS pseudo-element: input[type="password"].<class>::-ms-reveal
```



Range/Slider (<input type="range">)

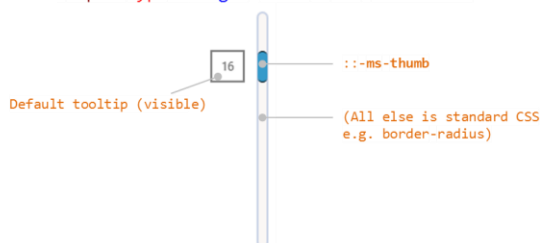
```
CSS pseudo-elements on input[type="range"].<class>
```

```
::-ms-ticks-before (top side)
::-ms-track (track area incl. ticks)
::-ms-ticks-after (bottom side)
```



```
::-ms-tooltip { display:none; } (only recognized style)
```

```
<input type="range" class="win-vertical">
```



```
::-ms-fill-lower (background-image)
::-ms-fill-upper (background-image)
```



```
::-ms-track (color
and background-color
set to transparent)
```

Combo/list box (<select>)



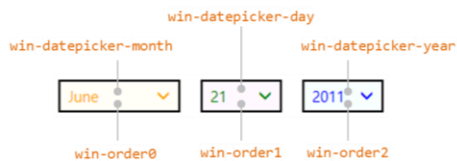
Note Though not shown here, you can also use the `-ms-scrollbar-*` styles for scrollbars that appear on pannable content in your app.

Styling Gallery: WinJS Controls

Similarly, here is a visual rundown of styling for WinJS controls, drawing again from the samples in the SDK: [HTML DatePicker and TimePicker controls](#), [HTML Rating control](#), [HTML ToggleSwitch control](#), and [HTML Tooltip control](#).

For the WinJS DatePicker and TimePicker, refer to styling for the HTML `select` element along with the `::-ms-value` and `::-ms-expand` pseudo-elements. I will note that the sample isn't totally comprehensive, so the visuals below highlight the finer points:

- `win-timepicker` and `win-datepicker` style the whole control (you override defaults)
- `win-datepicker-*` style individual parts (`display: none` will hide that part)
- `win-orderN` identifies the sub-element by position
- `Style { display: block; float: none }` on children for vertical layout



```
.win-datepicker [class^="win-datepicker"] {
  display: block;
  float: none;
}
```

```
.win-datepicker .win-datepicker-year {
  color: blue;
}

.win-datepicker .win-datepicker-date {
  color: green;
}

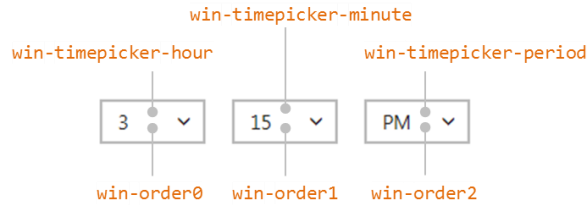
.win-datepicker .win-datepicker-month {
  color: orange;
}

.win-datepicker .win-datepicker-year::-ms-expand {
  color: red;
}

.win-datepicker .win-order0 {
  background-color: rgb(255, 255, 248);
}

.win-datepicker .win-order1 {
  background-color: rgb(255, 248, 255);
}

.win-datepicker .win-order2 {
  background-color: rgb(248, 255, 255);
}
```



The Rating control has states that can be styled in addition to its stars and the overall control. `win-*` classes identify these individually; combinations style all the variations as in this table:

Style Class	Part
<code>win-rating</code>	Styles the entire control.
<code>win-star</code>	Styles the control's stars generally.
<code>win-empty</code>	Styles the control's empty stars.
<code>win-full</code>	Styles the control's full stars.
<code>.win-star</code> Classes	State
<code>win-average</code>	Control is displaying an average rating (user has not selected a rating and the <code>averageRating</code> property is non-zero).
<code>win-disabled</code>	Control is disabled.
<code>win-tentative</code>	Control is displaying a tentative rating.
<code>win-user</code>	Control is displaying user-chosen rating.
Variation	Classes (selectors)
Average empty stars	<code>.win-star.win-average.win-empty</code>
Average full stars	<code>.win-star.win-average.win-full</code>
Disabled empty stars	<code>.win-star.win-disabled.win-empty</code>
Disabled full stars	<code>.win-star.win-disabled.win-full</code>
Tentative empty stars	<code>.win-star.win-tentative.win-empty</code>
Tentative full stars	<code>.win-star.win-tentative.win-full</code>
User empty stars	<code>.win-star.win-user.win-empty</code>
User full stars	<code>.win-star.win-user.win-full</code>

`.win-rating .win-star.win-user.win-full` (colors)



`.win-rating .win-star` (font size)



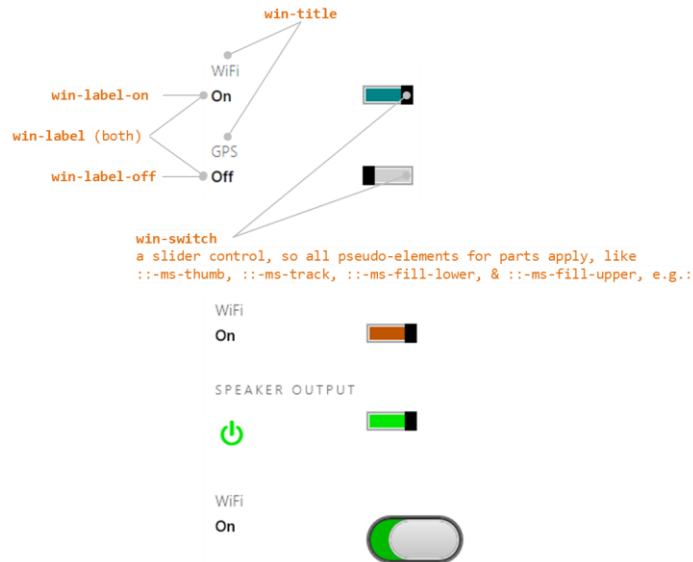
`class="win-small"`



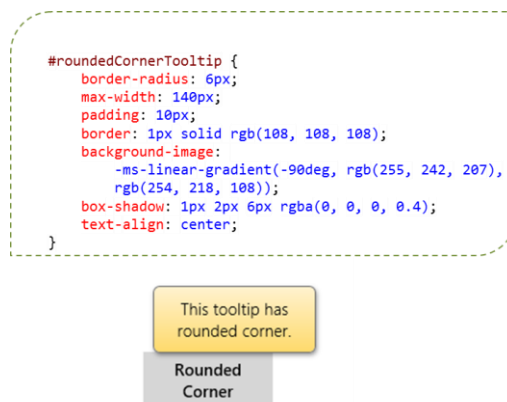
`.win-rating .win-star` (background image)

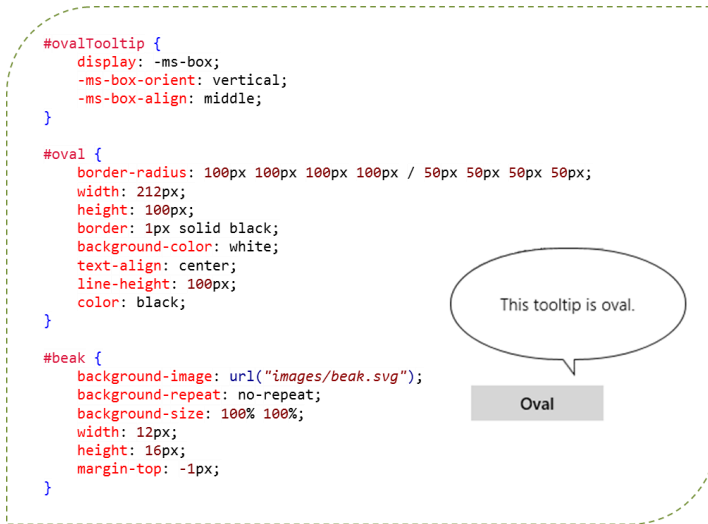


For the ToggleSwitch, `win-*` classes identify parts of the control; states are implicit. Note that the `win-switch` part is just an HTML slider control (`<input type="range">`), so you can utilize all the pseudo-elements for its parts.



And finally, for Tooltip, `win-tooltip` is a single class for the tooltip as a whole; the control can then contain any other HTML to which CSS applies using normal selectors:





Some Tips and Tricks

- In the current implementation, tooltips on a slider (`<input type="range">`) are always numerical values; there isn't a means to display other forms of text, such as *Low*, *Medium*, and *High*. For something like this, you could consider a `WinJS.UI.Rating` control with three values, using the `tooltipStrings` property to customize the tooltips.
- The `::-ms-tooltip` pseudo-selector for the slider affects only visibility (with `display: none`); it cannot be used to style the tooltip generally. This is useful to hide the default tooltips if you want to implement custom UI of your own.
- There are additional types of `input` controls (different values for the `type` attribute) that I haven't mentioned. This is because those types have no special behaviors and just render as a text box. Those that have been specifically identified might also just render as a text box, but they can affect, for example, what on-screen keyboard configuration is displayed on a touch device and also provide specific input validation (e.g., the number type only accepts digits).
- If you don't find `width` and `height` properties working for a control, try using `style.width` and `style.height` instead.
- You'll notice that there are two kinds of button controls: `<button>` and `<input type="button">`. They're visually the same, but the former is a block tag and can display HTML inside itself, whereas the latter is an inline tag that displays only text. A `button` also defaults to `<input type="submit">`, which has its own semantics, so you generally want to use `<button type="button">` to be sure.
- If a `WinJS.UI.Tooltip` is getting clipped, you can override the `max-width` style in the

`win-tooltip` class, which is set to 30em in the WinJS stylesheets. Again, peeking at the style in Blend's Style Rules tab is a quick way to see the defaults.

- The HTML5 `meter` element is not supported for WinRT apps.
- There's a default dotted outline for a control when it has the focus (tabbing to it with the keyboard or calling the `focus` method in JavaScript). To turn off this default rectangle for a control, use `<selector>:focus { outline: none; }` in CSS.
- WinRT apps can use the `window.getComputedStyle` method to obtain a `currentStyle` object that contains the applied styles for an element, or for a pseudo-element. This is very helpful, especially for debugging, because pseudo-elements like `::-ms-thumb` for an HTML slider control never appear in the DOM, so the styling is not accessible through the element's `style` property nor does it surface in tools like Blend. Here's an example of retrieving the background color style for a slider thumb:

```
var styles = window.getComputedStyle(document.getElementById("slider1"), "::-ms-thumb");
styles.getPropertyValue("background-color");
```

Custom Controls

As extensive as the HTML and WinJS controls are, there will always be something you wish the system provided but doesn't. "Is there a calendar control?" is a question I've often heard. "What about charting controls?" These clearly aren't included directly in Windows 8, and despite any wishing to the contrary, it means you or another third-party will need to create a custom control.

Fortunately, everything we've learned so far, especially about WinJS controls, applies to custom controls. In fact, WinJS controls are entirely implemented using the same model that you can use directly, and since you can look at the WinJS source code anytime you like, you already have a bunch of reference implementations available.

To go back to our earlier definition, a control is just declarative markup (creating elements in the DOM) plus applicable CSS plus methods, properties, and events accessible from JavaScript. To create such a control in the WinJS model, generally follow this pattern:

1. Define a namespace for your control(s) by using `WinJS.Namespace.define` to both provide a naming scope and to keep excess identifiers out of the global namespace. (Do *not* add controls to the WinJS namespace.) Remember that you can call `WinJS.Namespace.define` many times to add new members, so typically an app will just have a single namespace for all its custom controls.
2. Within that namespace, define the control constructor by using `WinJS.Class.define` (or `derive`), assigning the return value to the name you want to use in `data-win-control` attributes. That fully qualified name will be `<namespace>.<constructor>`.

3. Within the constructor (of the form `<constructor>(element, options)`):
 - a. You can recognize any set of options you want; these are arbitrary. Simply ignore any that you don't recognize.
 - b. If `element` is `null` or `undefined`, create a `div` to use in its place.
 - c. Assuming `element` is the root element containing the control, be sure to set `element.winControl=this` and `this.element=element` to match the WinJS pattern.
4. Within `WinJS.Class.define`, the second argument is an object containing your public methods and properties (those accessible through an instantiated control instance); the third argument is an object with static methods and properties (those accessible through the class name without needing to call `new`).
5. For your events, mix (`WinJS.Class.mix`) your class with the results from `WinJS.Utilities.createEventProperties(<events>)` where `<events>` is an array of your event names (without on prefixes). This will create `on<event>` properties in your class for each name in the list.
6. Also mix your class with `WinJS.UI.DOMEventMixin` to add standard implementations of `addEventListener`, `removeEventListener`, `dispatchEvent`, and `setOptions`.²⁷
7. In your implementation (markup and code), refer to classes that you define in a default stylesheet but that can be overridden by consumers of the control. Consider using existing `win-*` classes to align with general styling.
8. A typical best practice is to organize your custom controls in per-control folders that contain all the html, js, and css files for that control. Remember also that calls to `WinJS.Namespace.define` for the same namespace are additive, so you can populate a single namespace with controls that are defined in separate files.

You might consider using `WinJS.UI.Pages` if what you need is mostly a reusable block of HTML/CSS/JavaScript for which you don't necessarily need a bunch of methods, properties, and events. `WinJS.UI.Pages` is, in fact, implemented as a custom control. Along similar lines, if what you need is a reusable block of HTML in which you want to do run-time data binding, check out `WinJS.Binding.Template` (which we'll see toward the end of this chapter), which exists for that purpose. This isn't a control as we've been describing here—it doesn't support events, for instance—but might be exactly what you need.

It's also worth reminding you that everything in WinJS, like `WinJS.Class.define` and `WinJS.UI.DOMEventMixin` are just helpers for common patterns. You're not in any way required to use these, because in the end, custom controls are just elements in the DOM like any others and you can create and manage them however you like. The WinJS utilities just make most jobs cleaner and easier.

²⁷ Note that there is also a `WinJS.Utilities.eventMixin` that is similar (without `setOptions`) that is useful for noncontrol objects that won't be in the DOM but still want to fire events. The implementations here don't participate in DOM event bubbling/tunneling.

Custom Control Examples

To see these recommendations in action, here are a couple of examples. First is what Chris Tavares, one of the WinJS engineers who has been a tremendous help with this book, described as the “dumbest control you can imagine.” Yet it certainly shows the most basic structures:

```
WinJS.Namespace.define("AppControls", {
    HelloControl: function (element, options) {
        element.winControl = this;
        this.element = element;

        if (options.message) {
            element.innerText = options.message;
        }
    }
});
```

With this, you can then use the following markup so that `WinJS.UI.process/processAll` will instantiate an instance of the control (as an inline element because we’re using `span` as the root):

```
<span data-win-control="AppControls.HelloControl"
      data-win-options="{ message: 'Hello, World'}">
</span>
```

Note that the control definition code must be executed before `WinJS.UI.process/processAll` so that the constructor function named in `data-win-control` actually exists at that point.

For a more complete control, you can take a look at the [HTML SemanticZoom for custom controls sample](#) in the Windows SDK. My friend Kenichiro Tanaka of Microsoft Tokyo also created the calendar control shown in Figure 4-9 and provided in the `CalendarControl` example for this chapter.

Following the guidelines given earlier, this control is defined using `WinJS.Class.define` within a Controls namespace (calendar.js lines 4–10 shown here [with a comment line omitted]):

```
WinJS.Namespace.define("Controls", {
    Calendar : WinJS.Class.define(
        function (element, options) {
            this.element = element || document.createElement("div");
            this.element.className = "control-calendar";
            this.element.winControl = this;
```

The rest of the constructor (lines 12–63) builds up the child elements that define the control, making sure that each piece has a particular class name that, when scoped with the `control-calendar` class placed on the root element above, allows specific styling of the individual parts. The defaults for this are in `calendar.css`; specific overrides that differentiate the two controls in Figure 4-9 are in `default.css`.

Custom Calendar Control Demo



("year":2012,"month":5,"day":16) selected

Figure 4-9 Output of the Calendar Control demo sample.

Within the constructor you can also see that the control wires up its own event handlers for its child elements, such as the previous/next buttons and each date cell. In the latter case, clicking a cell uses `dispatchEvent` to raise a `dateselected` event from the overall control itself.

Lines 63–127 then define the members of the control. There are two internal methods, `_setClass` and `_update`, followed by two public methods, `nextMonth` and `prevMonth`, followed by three public properties, `year`, `month`, and `date`. Those properties can be set through the `data-win-options` string in markup or directly through the control object as we'll see in a moment.

At the end of `calendar.js` you'll see the two calls to `WinJS.Class.mix` to add properties for the events (there's only one here), and the standard DOM event methods like `addEventListener`, `removeEventListener`, and `dispatchEvent`, along with `setOptions`:

```
WinJS.Class.mix(Controls.Calendar, WinJS.Utilities.createEventProperties("dateselected"));
WinJS.Class.mix(Controls.Calendar, WinJS.UI.DOMEventMixin);
```

Very nice that adding all these details is so simple—thank you, WinJS!²⁸

Between `calendar.js` and `calendar.css` we have the definition of the control. In `default.html` and `default.js` we can then see how the control is used. In Figure 4-9, the control on the left is declared in markup and instantiated through the call to `WinJS.UI.processAll` in `default.js`.

```
<div id="calendar1" class="control-calendar" aria-label="Calendar 1"
    data-win-control="Controls.Calendar"
    data-win-options="{ year: 2012, month: 5, ondateselected: CalendarDemo.dateselected}">
</div>
```

²⁸ Technically speaking, `WinJS.Class.mix` accepts a variable number of arguments, so you can actually combine the two calls above into a single one.

You can see how we use the fully qualified name of the constructor as well as the event handler we're assigning to `ondateselected`. But remember that functions referenced in markup like this have to be marked for strict processing. The constructor is automatically marked through `WinJS.Class.define`, but the event handler needs extra treatment: we place the function in a namespace (to make it globally visible) and use `WinJS.UI.eventHandler` to do the marking:

```
WinJS.Namespace.define("CalendarDemo", {
    dateselectd: WinJS.UI.eventHandler(function (e) {
        document.getElementById("message").innerText = JSON.stringify(e.detail) + " selected";
    })
});
```

Again, if you forget to mark the function in this way, the control won't be instantiated at all. (Remove the `WinJS.UI.eventHandler` wrapper to see this.)

To demonstrate creating a control outside of markup, the control on the right of Figure 4-9 is created as follows, within the `calendar2` `div`:

```
//Since we're creating this calendar in code, we're independent of WinJS.UI.processAll.
var element = document.getElementById("calendar2");

//Since we're providing an element, this will be automatically added to the DOM
var calendar2 = new Controls.Calendar(element);

//Since this handler is not part of markup processing, it doesn't need to be marked
calendar2.ondateselected = function (e) {
    document.getElementById("message").innerText = JSON.stringify(e.detail) + " selected";
}
```

There you have it!

Note For a control you really intend to share with others, you'll want to include the necessary comments that provide metadata for IntelliSense. See the "Sidebar: Helping Out IntelliSense" in Chapter 3 for more details.

Custom Controls in Blend

Blend is an excellent design tool for working with controls directly on the artboard, so you might be wondering how custom controls integrate into that story.

First, since custom controls are just elements in the DOM, Blend works with them like all other parts of the DOM. Try loading the Calendar Control Demo into Blend to see for yourself.

Next, a control can determine if it's running inside Blend's design mode if the `Windows.ApplicationModel.DesignMode.designModeEnabled` property is `true`. One place where this is very useful is when handling resource strings. We won't cover resources in full until Chapter 17, but it's important to know here that resource lookup, through `Windows.ApplicationModel.Resources.ResourceLoader` in Blend's design mode, doesn't work the same as when the app is actually running for real. To be blunt, it doesn't work at all and throws exceptions! So you can use the design-mode flag

to just provide a suitable default instead of doing the lookup.

For example, one of the early partners I worked with had a method to retrieve a localized URI to their back-end services, which was failing in design mode. Using the design mode flag, then, we just had to change the code to look like this:

```
WinJS.Namespace.define("App.Localization", {
    getBaseUrl: function () {
        if (Windows.ApplicationModel.DesignMode.designModeEnabled) {
            return "www.default-base-service.com";
        } else {
            var resources = new Windows.ApplicationModel.Resources.ResourceLoader();
            var baseUrl = resources.getString("baseUrl");
            return baseUrl;
        }
    }
});
```

Finally, it is possible to have custom controls show up in the Assets tab alongside the HTML elements and the WinJS controls. For this you'll first need an [OpenAjax Metadata XML \(OAM\) file](#) that provides all the necessary information for the control, and you already have plenty of references to draw from. Take a look on your machine under *Program Files (x86)\Microsoft Visual Studio 11.0\Blend\Intrinsics* in some of the subfolders and you'll find plenty of *_oam.xml files. The ones for WinJS controls are in an even more obscure location: *Program Files (x86)\Microsoft SDKs\Windows\v8.0\ExtensionSDKs\Microsoft.WinJS.1.0.RC\1.0\DesignTime\CommonConfiguration\Neutral\Microsoft.WinJS.1.0.RC\js\metadata*. In both places you'll also find plenty of examples of the 12x12 and 16x16 icons you'll want for your control.

If you look in the controls/calendar folder of the CalendarControl sample with this chapter, you'll find calendar_oam.xml and two icons alongside the .js and .css files. The OAM file (that must have a filename ending in _oam.xml) tells Blend how to display the control in its Assets panel and what code it should insert when you drag and drop a control into an HTML file. Here are the contents of that file:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Use underscores or periods in the id and name, not spaces. -->
<widget version="1.0"
    spec="1.0"
    id="http://www.kraigbrockschmidt.com/scehmas/ProgrammingWin8_JS/Controls/Calendar"
    name="ProgWin8_JS.Controls.Calendar"
    xmlns="http://openajax.org/metadata">

    <author name="Kenichiro Tanaka" />

    <!-- title provides the name that appears in Blend's Assets panel
         (otherwise it uses the widget.name). -->
    <title type="text/plain"><![CDATA[Calendar Control]]></title>

    <!-- description provides the tooltip fir Assets panel. -->
    <description type="text/plain"><![CDATA[A single month calendar]]></description>

    <!-- icons (12x12 and 16x16 provide the small icon next to the control
```

```

        in the Assets panel. -->
<icons>
    <icon src="calendar.16x16.png" width="16" height="16" />
    <icon src="calendar.12x12.png" width="12" height="12" />
</icons>

<!-- This element describes what gets inserted into the .html file;
comment out anything that's not needed -->
<requires>
    <!-- The control's code -->
    <require type="javascript" src="calendar.js" />

    <!-- The control's stylesheet -->
    <require type="css" src="calendar.css" />

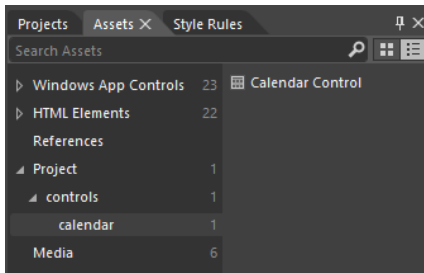
    <!-- Any inline script for the document head -->
    <require type="javascript"><![CDATA[WinJS.UI.processAll();]]></require>

    <!-- Inline CSS for the style block in the document head -->
    <!--<require type="css"><![CDATA[.control-calendar{}]]></require>-->
</requires>

<!-- What to insert in the body for the control; be sure this is valid HTML
or Blend won't allow insertion -->
<content>
    <![CDATA[
        <div class="control-calendar" data-win-control="Controls.Calendar"
            data-win-options="{ year: 2012, month: 6 }"></div>
    ]]>
</content>
</widget>

```

When you add all five files into a project in Blend, you'll see the control's icon and title in the Assets tab (and hovering over the control shows the tooltip):



If you drag and drop that control onto an HTML page, you'll then see the different bits added in:

```

<!DOCTYPE html>
<html>
<head>
    <!-- ... -->
    <script src="calendar.js" type="text/javascript"></script>
    <link href="calendar.css" rel="stylesheet" type="text/css">

```

```

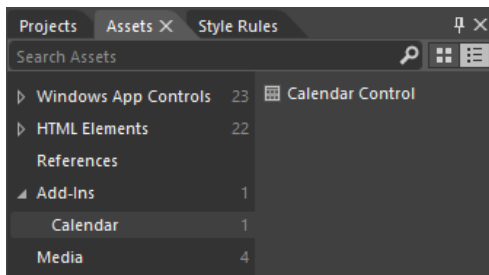
</head>
<body>
  <div class="control-calendar" data-win-control="Controls.Calendar"
    data-win-options="{month:6, year:2012}"></div>
</body>
</html>

```

Oh! What happened to the `WinJS.UI.processAll()` call? It just so happens that Blend singles out this piece of code to check if it's already being called somewhere in the loaded script. If it is (as is typical with the project templates), Blend doesn't repeat it. If it does include it, or if you specify other code here, Blend will insert it in a `<script>` tag in the header.

Also, errors in your OAM file will convince Blend that it shouldn't insert the control at all, so you'll need to fix those errors. When making changes, Blend won't reload the metadata unless you reload the project or rename the OAM file (preserving the `_oam.xml` part). I found the latter is much easier, as Blend doesn't care what the rest of the filename looks like. In this renaming process too, if you find that the control disappeared from the Assets panel, it means you have an error in the OAM XML structure itself, such as attribute values containing invalid characters. For this you'll need to do some trial and error, and of course you can refer to all the OAM files already on your machine for details.

You can also make your control available to all projects in Blend. To do this, go to *Program Files (x86)\Microsoft Visual Studio 11.0\Blend*, create a folder called *Addins* if one doesn't exist, create a subfolder therein for your control (using a reasonably unique name), and copy all your control assets there. When you restart Blend, you'll see the control listed under Addins in the Assets tab:



This would be appropriate if you create custom controls for other developers to use; your desktop installation program would simply place your assets in the Addins folder. As for using such a control, when you drag and drop the control to an HTML file, its required assets (but not the icons nor the OAM file) are copied to the project into the root folder. You can then move them around however you like, patching up the file references, of course.

Data Binding

As I mentioned in the introduction to this chapter, the subject of data binding is closely related to controls because it's how you create relationships between properties of data objects and properties of controls (including styles). This way, controls reflect what's happening in the data, which is often exactly

what you want to accomplish in your user experience.

I want to start this discussion with a review of data binding in general, for you may be familiar with the concept to some extent, as I was, but unclear on a number of the details. At times, in fact, especially if you're talking to someone who has been working with it for years, data binding seems to become shrouded in some kind of impenetrable mystique. I don't at all count myself among such initiates, so I'll try to express the concepts in prosaic terms.

The general idea of data binding is again to connect or "bind" properties of two different objects together, typically a data object and a UI object, which we can generically refer to as a source and a target. A key here is that data binding generally happens between *properties*, not objects.

The binding can also involve converting values from one type into another, such as converting a set of separate source properties into a single string as suitable for the target. It's also possible to have multiple target objects bound to the same source object or one target bound to multiple source objects. This flexibility is exactly why the subject can become somewhat nebulous, because there are so many possibilities! Still, for most scenarios, we can keep the story simple.

A common data-binding scenario is shown in Figure 4-10, where we have specific properties of two UI elements, a `span` and an `img`, bound to properties of a data object. There are three bindings here: (1) the `span.innerText` property is bound to the `source.name` property; (2) the `img.src` property is bound to the `source.photoURL` property; and (3) the `span.style.color` property is bound to the output of a converter function that changes the `source.userType` property into a color.

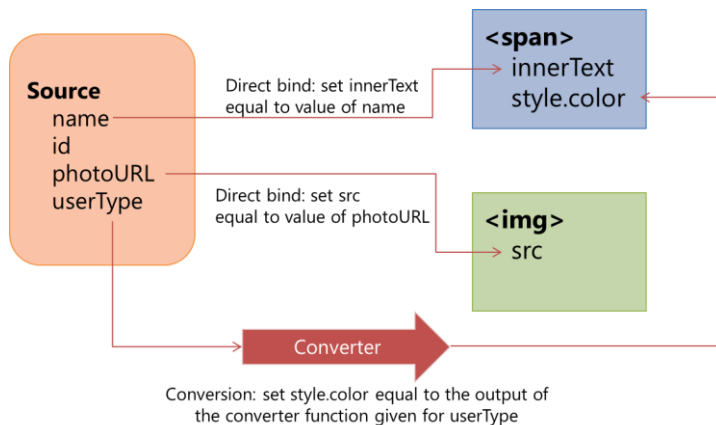
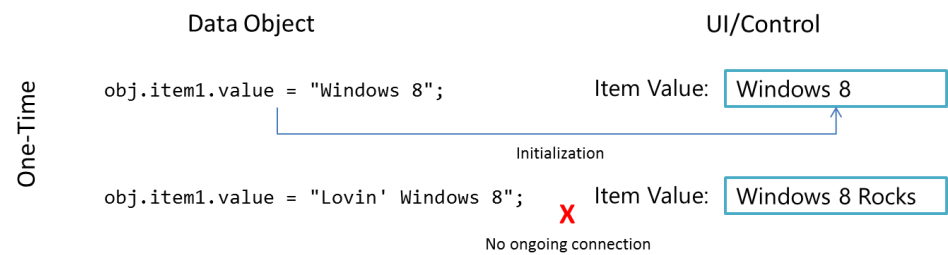


Figure 4-10 A common data-binding scenario between a source data object and two target UI elements, involving two direct bindings and one binding with a conversion function.

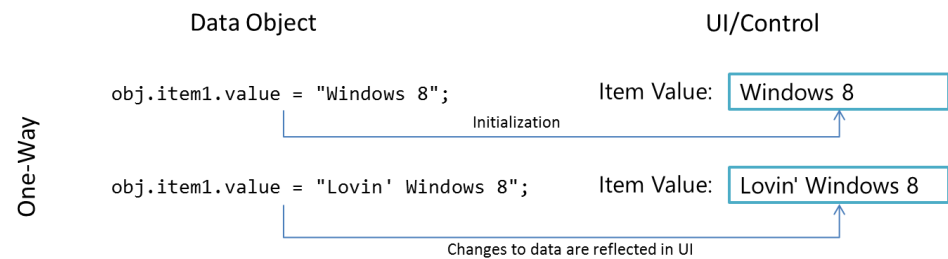
How these bindings actually behave at run time then depends on the particular *direction* of each binding, which can be one of the following:

One-time: the value of the source property (possibly with conversion) is copied to the target property at some point, after which there is no further relationship. This is what you automatically do

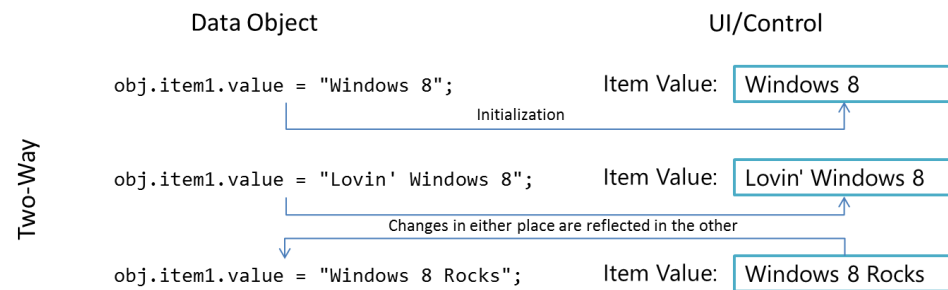
when passing variables to control constructors, for instance, or simply assigning target property values using source properties. (What’s useful here is to have a declarative means to make such assignments directly in element attributes.)



One-way: the target object listens for change events on bound source properties so that it can update itself with new values. This is typically used to update a UI element in response to underlying changes in the data. Changes within the target element (like a UI control), however, are not reflected back to the data itself (but can be sent elsewhere as with form submission, which could in turn update the data through another channel).



Two-way: essentially one-way binding in both directions, as the source object also listens to change events from the target object. Changes made within a UI element like a text box are thus saved back in the bound source property, just as changes to the data source property update the UI element. Obviously, there must be some means to not get stuck in an infinite loop; typically, both objects avoid firing another change event if the new value is the same as the existing one.



Data Binding in WinJS

Now that we've seen what data binding is all about, we can see how they can be implemented within a Windows 8 app. If you like, you can create whatever scheme you want for data binding or use a third-party JavaScript library for the job: it's just about connecting properties of source objects with properties of target objects.

Now, if you're anything like a number of my paternal ancestors, who seemed to wholly despise relying on anyone to do anything they could do themselves (like drilling wells, mining coal, and manufacturing engine parts), you may very well be content with engineering your own data-binding solution. But if you have a more tempered nature like I do (thanks to my mother's side), I'm delighted when someone is thoughtful enough to create a solution for me. Thus my gratitude goes out to the WinJS team who, knowing of the common need for data binding, created the `WinJS.Binding` API. This supports one-time and one-way binding, both declaratively and procedurally, along with converter functions. At present, WinJS does not provide for two-way binding, but such structures aren't difficult to set up in code, as we'll see.

Within the WinJS structures, multiple target elements can be bound to a single data source. `WinJS.Binding`, in fact, provides for what are called *templates*, basically collections of target elements that are together bound to the same data source. Though we don't recommend it, it's possible to bind a single target element to multiple sources, but this gets tricky to manage properly. A better approach in such cases is to wrap those separate sources into a single object and bind to that instead.

The best way, now, to understand `WinJS.Binding` is to first see look at how we'd write our own binding code and then see the solution that WinJS offers. For these examples, we'll use the same scenario as shown in Figure 4-10, where we have a source object bound to two separate UI elements, with one converter that changes a source property into a color.

One-Time Binding

One-time binding, as mentioned before, is essentially what you do whenever you just assign values to properties of an element. So, given this HTML:

```
<!-- Markup: the UI elements we'll bind to a data object -->
<section id="loginDisplay1">
  <p>You are logged in as <span id="loginName1"></span></p>
  <img id="photo1"></img>
</section>
```

and the following data source object:

```
var login1 = { name: "liam", id: "12345678",
  photoURL: "http://www.kraigbrockschmidt.com/images/Liam07.png", userType: "kid"};
```

we can bind as follows, also using a converter function in the process:

```
//"Binding" is done one property at a time, with converter functions just called directly
var name = document.getElementById("loginName1");
name.innerText = login1.name;
```

```

name.style.color = userTypeToColor1(login1.userType);
document.getElementById("photo1").src = login1.photoURL;

function userTypeToColor1(type) {
    return type == "kid" ? "Orange" : "Black";
}

```

This gives the following result, in which I shamelessly publish a picture of my kid:



The code for this can be found in Test 1 of the BindingTests example for this chapter. With WinJS we can accomplish the same thing by using a declarative syntax and a processing function. In markup, we use the attribute `data-win-bind` to map target properties of the containing element to properties of the source object that is given to the processing function, `WinJS.Binding.processAll`.

The value of `data-win-bind` is a string of property pairs. Each pair's syntax is `<target property> : <source property> [<converter>]` where the converter is optional. Each property identifier can use dot notation as needed, and property pairs are separated by a semicolon as shown in the HTML:

```

<section id="loginDisplay2">
    <p>You are logged in as
        <span id="loginName2"
            data-win-bind="innerText: name; style.color: userType Tests.userTypeToColor">
        </span>
    </p>
    <img id="photo2" data-win-bind="src: photoURL"/>
</section>

```

Note that array lookup on the source property using `[]`'s is not supported, though a converter could do that. On the target, if that object has a JavaScript property that you want to refer to using a hyphenated identifier, you can use the following syntax:

```

<span data-win-bind="this['funky-property']: source"></span>

```

A similar syntax is necessary for data-binding target *attributes*, such as the `aria-*` attributes for accessibility. Because these are not JavaScript properties, a special converter (or initializer as it is more properly called) named `WinJS.Binding.setAttribute` is needed:

```

<label data-win-bind="this['aria-label']: title WinJS.Binding.setAttribute"></label>

```

Also see `WinJS.Binding.setAttributeOneTime` for one-time binding for attributes.

Sidebar: Data-Binding Properties of WinJS Controls

When targeting properties on a WinJS control and not its root (containing) element, the target property names should begin with `winControl`. Otherwise you'll be binding to nonexistent properties on the root element. When using `winControl`, the bound property serves the same purpose as specifying a fixed value in `data-win-options`. For example, the markup used earlier in the “Example: WinJS.UI.Rating Control” section could use data binding for its `averageRating` and `userRating` properties as follows (assuming `myData` is an appropriate source):

```
<div id="rating1" data-win-control="WinJS.UI.Rating"
    data-win-options="{onChange: changeRating}"
    data-win-bind="{winControl.averageRating: myData.average,
        winControl.userRating: myData.rating}">
</div>
```

Anyway, assuming we have a data source as before:

```
var login2 = { name: "liamb", id: "12345678",
    photoURL: "http://www.kraigbrockschmidt.com/images/Liam07.png", userType: "kid"};
```

We convert the markup to actual bindings using `WinJS.Binding.processAll`:

```
//processAll scans the element's tree for data-win-bind, using given object as data context
WinJS.Binding.processAll(document.getElementById("loginDisplay2"), login2);
```

This code, `Test2` in the example, produces the same result as `Test 1`. The one added bit here is that we need to define the converter function so that it's globally accessible and marked for processing. This can be accomplished with a namespace that contains a function (actually called an initializer, as we'll discuss in the “Binding Initializers” section near the end of this chapter) created by `WinJS.Binding.converter`:

```
//Use a namespace to export function from the current module so WinJS.Binding can find it
WinJS.Namespace.define("Tests", {
    userTypeToColor: WinJS.Binding.converter(function (type) {
        return type == "kid" ? "Orange" : "Black";
    })
});
```

As with control constructors defined with `WinJS.Class.define`, `WinJS.Binding.converter` automatically marks the functions it returns as safe for processing.

We could also put the data source object and applicable converters within the same namespace.²⁹ For example (in `Test 3`), if we placed our `login` data object and the `userTypeToColor` function in a `LoginData` namespace, the markup and code would look like this:

²⁹ More commonly, converters would be part of a namespace in which applicable UI elements are defined, because they're more specific to the UI than to a data source.

```
<span id="loginName3"
  data-win-bind="innerText: name; style.color: userType LoginData.userTypeToColor">
</span>
```

```
WinJS.Binding.processAll(document.getElementById("loginDisplay3"), LoginData.login);
```

```
WinJS.Namespace.define("LoginData", {
  login : {
    name: "liamb", id: "12345678",
    photoURL: "http://www.kraigbrockschmidt.com/images/Liam07.png",
    userType: "kid"
  },

  userTypeToColor: WinJS.Binding.converter(function (type) {
    return type == "kid" ? "Orange" : "Black";
  })
});
```

In summary, for one-time binding `WinJS.Binding` simply gives you a declarative syntax to do exactly what you'd do in code, with a lot less code. Because it's all just some custom markup and a processing function, there's no magic here, though such useful utilities are magical in their own way! In fact, the code here is really just one-way binding without having the source fire any change events. We'll see how to do that with `WinJS.Binding.as` in a moment after a couple more notes.

First, `WinJS.Binding.processAll` is actually an async function that returns a promise. Any completed handler given to its `done` method will be called when the processing is finished, if you have additional code that's depending on that state. Second, you can call `WinJS.Binding.processAll` more than once on the same target element, specifying a different source object (data context) each time. This won't replace any existing bindings, mind you—it just adds new ones, meaning that you could end up binding the same target property to more than one source, which could become a big mess. So again, a better approach is to combine those sources into a single object and bind to that, using dot notation to identify nested properties.

One-Way Binding

The goal for one-way binding is, again, to update a target property, typically in a UI control, when the bound source property changes. That is, one-way binding means to effectively repeat the one-time binding process whenever the source property changes.

In the code we saw above, if we changed `login.name` after calling `WinJS.Binding.processAll`, nothing will happen in the output controls. So how can we automatically update the output?

Generally speaking, this requires that the data source maintains a list of *bindings*, where each binding could describe a source property, a target property, and a converter function. The data source would also need to provide methods to manage that list, like *addBinding*, *removeBinding*, and so forth. Thirdly, whenever one of its bindable (or *observable*) properties changes it goes through its list of bindings and updates any affected target property accordingly.

These requirements are quite generic; you can imagine that their implementation would pretty much join the ranks of classic boilerplate code. So, of course, WinJS provides just such an implementation! In this context, sources are called *observable objects*, and the function `WinJS.Binding.as` wraps any arbitrary object with just such a structure. (It's a no-op for nonobjects.) Conversely, `WinJS.Binding.unwrap` removes that structure if there's a need. Furthermore, `WinJS.Binding.define` creates a constructor for observable objects around a set of properties (described by a kind of empty object that just has property names). Such a constructor allows you to instantiate source objects dynamically, as when processing data retrieved from an online service.

So let's see some code. Going back to the last example above (Test 3), any time before or after `WinJS.Binding.processAll` we can take the `LoginData.login` object and make it observable as follows:

```
var loginObservable = WinJS.Binding.as(LoginData.login)
```

This is actually all we need to do—with everything else the same as before, we can now change a bound property within the `loginObservable` object:

```
loginObservable.name = "liambro";
```

This will update the target property:



Here's how we'd then create and use a reusable class for an observable object (Test 4 in the BindingTests example). Notice the object we pass to `WinJS.Binding.define` contains property names, but no values (they'll be ignored):

```
WinJS.Namespace.define("LoginData", {  
    //...  
  
    //LoginClass becomes a constructor for bindable objects with the specified properties  
    LoginClass: WinJS.Binding.define({name: "", id: "", photoURL: "", userType: "" }},  
});
```

With that in place, we can create an instance of that class, initializing desired properties. In this example, we're using a different picture and leaving `userType` uninitialized:

```
var login4 = new LoginData.LoginClass({ name: "liamb",  
    photoURL: "http://www.kraigbrockschmidt.com/images/Liam08.jpg" });
```

Binding to this `login` object, we'd see that the username initially comes out black.

```
//Do the binding (initial color of name would be black)
WinJS.Binding.processAll(document.getElementById("loginDisplay"), login4);
```

Updating the `userType` property in the source (as below) would then cause an update the color of the target property, which happens through the converter automatically:

```
login4.userType = "kid";
```



Implementing Two-Way Binding

To implement two-way binding, the process is straightforward:

1. Add listeners to the appropriate UI element events that relate to bound data source properties.
2. Within those handlers, update the data source properties.

The data source should be smart enough to know when the new value of the property is already the same as the target property, in which case it shouldn't try to update the target lest you get caught in a loop. The observable object code that WinJS provides does this type of check for you.

To see an example of this, refer to the [Declarative Binding sample](#) in the SDK, which listens for the `change` event on text boxes and updates values in its source accordingly.

Additional Binding Features

If you take a look at the [WinJS.Binding reference](#) in the documentation, you'll see a number of other goodies in the namespace. Let me briefly outline the purpose of these.

If you already have a defined class (from `WinJS.Class.define`) and want to make it observable, use `WinJS.Class.mix` as follows:

```
var MyObservableClass = WinJS.Class.mix(MyClass, WinJS.Binding.mixin,
    WinJS.Binding.expandProperties(MyClass));
```

`WinJS.Binding.mixin` here contains a standard implementation of the binding functions that WinJS expects. `WinJS.Binding.expandProperties` creates an object whose properties match those in the given object (the same names), with each one wrapped in the proper structure for binding. Clearly, this type of operation is useful only when doing a mix, and it's exactly what `WinJS.Binding.define` does with the oddball, no-values object we give to it.

If you remember from a previous section, one of the requirements for an observable object is that it contains methods to manage a list of bindings. An implementation of such methods is contained in the `WinJS.Binding.observableMixin` object. Its methods are:

- `bind` Saves a binding (property name and a function to invoke on change).
- `unbind` Removes a binding created by `bind`.
- `Notify` Goes through the bindings for a property and invokes the functions associated with it. This is where WinJS checks that the old and new values are actually different and where it also handles cases where an update for the same target is already in progress.

Building on this is yet another mixin, `WinJS.Binding.dynamicObservableMixin` (which is what `WinJS.Binding.mixin` is), which adds methods for managing source properties as well:

- `setProperty` Updates a property value and notifies listeners if the value changed.
- `updateProperty` Like `setProperty`, but returns a promise that completes when all listeners have been notified (the result in the promise is the new property value).
- `getProperty` Retrieves a property value as an observable object itself, which makes it possible to bind within nested object structures (`obj1.obj2.prop3`, etc.).
- `addProperty` Adds a new property to the object that is automatically enabled for binding.
- `removeProperty` Removes a property altogether from the object.

Why would you want all of these? Well, there are some creative uses. You can call `WinJS.Binding.bind`, for example, directly on any observable source when you want to hook up another function to a source property. This is like adding event listeners for source property changes, and you can have as many listeners as you like. This is helpful for wiring up two-way binding, and it doesn't in any way have to be related to manipulating UI. The function just gets called on the property change. This could be used to autosync a back-end service with the source object.

The Declarative Binding sample in the SDK (again, found [here](#)) also shows calling `bind` with an object as the second parameter, a form that allows for binding to nested members of the source. The syntax looks like this: `bind(rootObject, { property: { sub-property: function(value) { ... } } })`—whatever matches the source object. With such an object in the second parameter, `bind` will make sure to invoke all the functions assigned to the nested properties. In such a case, the return value of `bind` is an object with a `cancel` method that will clear out this complex binding.

The `notify` method, for its part, is something you can call directly to trigger notifications. This is useful with additional bindings that don't necessarily depend on the values themselves, just the fact that they changed. The major use case here is to implement computed properties—ones that change in response to another property value changing.

The system here also has some intelligent handling of multiple changes to the same source property. After the initial binding, further change notifications are asynchronous and multiple pending changes to the same property are coalesced. So, if in our example we made several changes to the name property in quick succession:

```
login.name = "Kenichiro";  
login.name = "Josh";  
login.name = "Chris";
```

only one notification for the last value would be sent and that would be the value that shows up in bound targets.

Finally, here are a few more functions hanging off `WinJS.Binding`:

- **oneTime** A function that just loops through the given target (destination) properties and sets them to the value of the associated source properties. This function can be used for true one-time bindings, as is necessary when binding to WinRT objects. It can also be used directly as an initializer within `data-win-bind` if the source is a WinRT object.
- **defaultBind** A function that does the same as **oneTime** but establishes one-way binding between all the given properties. This also serves as the default initializer for all relationships in `data-win-bind` when specific initializer isn't specified.
- **declarativeBind** The actual implementation of **processAll**. (The two are identical.) In addition to the common parameters (the root target element and the data context), it also accepts a **skipRoot** parameter (if true, processing does not bind properties on the root element, only its children, which is useful for template objects) and **bindingCache** (an optimization for holding the results of parsing the data-win-bind expression when processing template objects).

Binding Initializers

In our earlier examples we saw some uses of converter functions that turn some bit of source data into whatever a target property expects. But the function you specify in `data-win-bind` is more properly called an *initializer* because in truth it's only ever called once.

Say what? Aren't converters used whenever a bound source property gets copied to the target? Well, yes, but we're actually talking about two different functions here. Look carefully at the code structure for the `userTypeToColor` function we used earlier:

```
userTypeToColor: WinJS.Binding.converter(function (type) {  
    return type == "kid" ? "Orange" : "Black";  
})
```

The `userTypeToColor` function itself is an *initializer*. When it's called—once and only once—its *return value* from `WinJS.Binding.converter` is the *converter* that will then be used for each property update. That is, the real converter function is not `userTypeToColor`—it's actually a structure that wraps

the anonymous function given to `WinJS.Binding.converter`.

Under the covers, `WinJS.Binding.converter` is actually using `bind` to set up relationships between source and target properties, and it inserts your anonymous conversion function into those relationships. Fortunately, you generally don't have to deal with this complexity and can just provide that conversion function, as shown above.

Still, if you want a raw example, check out the Declarative Binding sample again, as it shows how to create a converter for complex objects directly in code without using `WinJS.Binding.converter`. In this case, that function needs to be marked as safe for processing if it's referenced in markup. Another function, `WinJS.Binding.initializer` exists for that exact purpose; the return value of `WinJS.Binding.converter` passes through that same method before it comes back to your app.

Binding Templates and Lists

Did you think we'd exhausted `WinJS.Binding` yet? Well, my friend, not quite! There are two more pieces to this rich API that lead us directly into the next chapter. (Now you know the real reason I put this entire section where I did!). The first is `WinJS.Binding.List`, a bindable *collection* data source that—not surprisingly—is very useful when working with collection controls.

`WinJS.Binding.Template` is also a unique kind of custom control. In usage, as you can again see in the Declarative Binding sample, you declare an element (typically a `div`) with `data-win-control = "WinJS.Binding.Template"`. In that same markup, you specify the template's contents as child elements, any of which can have `data-win-bind` attributes. What's unique is that when `WinJS.UI.process` or `processAll` hits this markup, it instantiates the template and actually pulls everything but the root element *out* of the DOM entirely. So what good is it then?

Well, once that template exists, anyone can call its `render` method to create a copy of that template within some other element, using some data context to process any `data-win-bind` attributes therein (typically skipping the root element itself, hence that `skipRoot` parameter in the `WinJS.Binding.declarativeBind` method). Furthermore, rendering a template multiple times into the same element creates multiple siblings, each of which can have a different data source.

Ah ha! Now you can start to see how this all makes perfect sense for collection controls and collection data sources. Given a collection data source and a template, you can iterate over that source and render a copy of the template for each source item into some other element. Add a little navigation or layout within that containing element and voila! You have the beginnings of what we know as the `WinJS.UI.FlipView` and `WinJS.UI.ListView` controls, as we'll explore next.

What We've Just Learned

- The overall control model for HTML and WinJS controls, where every control consists of declarative markup, applicable CSS, and methods, properties, and events accessible through JavaScript.

- Standard HTML controls have dedicated markup; WinJS controls use `data-win-control` attributes, which are processed using `WinJS.UI.process` or `WinJS.UI.processAll`.
- Both types of controls can also be instantiated programmatically using `new` and the appropriate constructor, such as `Button` or `WinJS.UI.Rating`.
- All controls have various options that can be used to initialize them. These are given as specific attributes in HTML controls and within the `data-win-options` attribute for WinJS controls.
- All controls have standard styling as defined in the WinJS stylesheets: `ui-light.css` and `ui-dark.css`. Those styles can be overridden as desired, and some style classes, like `win-backbutton`, are used to style a standard HTML control to look like a Windows-specific control.
- Windows 8 apps have rich styling capabilities for both HTML and WinJS controls alike. For HTML controls, `-ms-*`-prefixed pseudo-selectors allow you to target specific pieces of those controls. For WinJS controls, specific parts are styled using `win-*` classes that you can override.
- Custom controls are implemented in the same way WinJS controls are, and WinJS provides standard implementations of methods like `addEventListener`. Custom controls can also be shown in Blend's Assets panel either for a single project or for all projects.
- WinJS provides declarative data-binding capabilities for one-time and one-way binding, which can employ conversion functions. It even provides the capability to create an observable (one-way bindable) data source from any other object.
- WinJS also provides support for bindable collections and templates that can be repeatedly rendered for different source objects into the same containing element, which is the basis for collection controls.

Chapter 5

Collections and Collection Controls

It's a safe bet to say that wherever you are, right now, you're probably surrounded by quite a number of collections. This book you're reading is a collection of chapters, and chapters are a collection of pages. Those pages are collections of paragraphs, which are collections of words, which are collections of letters, which are (assuming you're reading this electronically) collections of pixels. On and on....

Your body, too, has collections on many levels, which is very much what one studies in college-level anatomy courses. Looking around my office and my home, I see even more collections: a book shelf with books; scrapbooks with pages and pages with pictures; cabinets with cans, boxes, and bins of food; my son's innumerable toys; the DVD case...even the forest outside is a collection of trees and bushes, which then have branches, which then have leaves. On and on....

We look at these things as collections because we've learned how to generalize specific instances of unique things—like leaves or pages or my son's innumerable toys—into categories or groups. This gives us powerful means to organize and manage those things (except for the clothes in my closet, as my wife will attest). And just as the physical world around us is very much made of collections, the digital world that we use to represent the physical is naturally full of collections as well. Thus programming languages like JavaScript have constructs like arrays to organize and manage collection data, and environments like Windows 8 provide collection controls through which we can visualize and manipulate that data.

In this chapter we'll turn our attention to the two collection controls provided by WinJS: the FlipView, which shows one item from a collection at a time, and the ListView, which shows many items in different arrangements. As you might expect, the ListView is the richer of the two, and as it's really the centerpiece of many app designs, we'll be spending the bulk of this chapter exploring its depths, along with the concept and implementation of *semantic zoom* (another control, in fact).

As both collection controls can handle items of arbitrary complexity (both in terms of data and presentation, unlike the simple HTML listbox and combobox controls), as well as an arbitrary number of items, they naturally build on the foundations of data binding and template controls we just saw at the end of Chapter 4, "Controls, Control Styling, and Data Binding." They also have a close relationship to collection data sources, which we'll specifically examine as well, and their own styling and behavioral considerations.

But let's not exhaust our minds here at the outset of this chapter with theory or architectural intricacies! Instead, let's just jump into some code to explore the core aspects of both controls.

Collection Control Basics

To seek the basics of the collection controls, we'll first look at the FlipView which will introduce us to item templates and data sources. We'll then see how these also apply to the ListView control, then look at grouping items within a ListView..

Quickstart #1: The HTML FlipView Control Sample

As shown in Figure 5-1, the [FlipView sample](#) in the Windows SDK is both a great piece of reference code for this control and a great visual tool through which to explore the control itself. (I'm also extremely grateful that I've not had to write such samples for this book!) For the purposes of this Quickstart, let's just look at the first scenario of populating the control from a simple data source and using a template for rendering the items, as these mechanisms are shared with the ListView. We'll come back to the other FlipView scenarios later in the chapter.

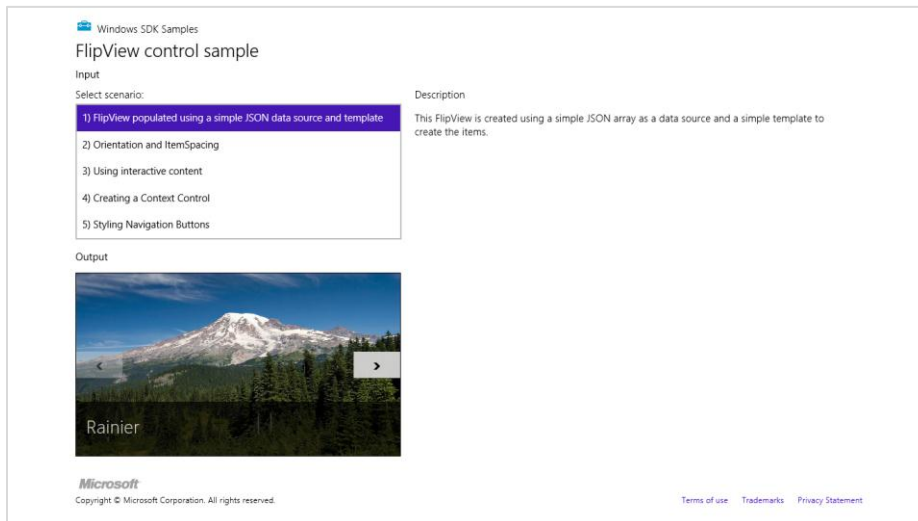


Figure 5-1 The SDK's FlipView control sample; the FlipView is the control displaying the picture.

As FlipView is a WinJS control, whose constructor is `WinJS.UI.FlipView`, we declare it in markup with `data-win-control` and `data-win-options` attributes (see `html/simpleFlipview.html`):

```
<div id="simple_FlipView" class="flipView" data-win-control="WinJS.UI.FlipView"
    data-win-options="{ itemDataSource: DefaultData.bindingList.dataSource,
        itemTemplate: simple_ItemTemplate }">
</div>
```

And of course, `WinJS.UI.processAll` is called in the page-loading process to instantiate the control. In the FlipView's options we can immediately see the two critical pieces to make the control work: a data source that provides the goods for each item and a template to render them.

If you were paying attention at the end of Chapter 4, you’ve probably guessed that the template is an instance of `WinJS.Binding.Template`. And you’re right! That piece of markup, in fact, comes just before the control declaration in `simpleFlipview.html`.

```
<div id="simple_ItemTemplate" data-win-control="WinJS.Binding.Template" style="display: none">
  <div class="overlaidItemTemplate">
    <img class="image" data-win-bind="src: picture; alt: title" />
    <div class="overlay">
      <h2 class="ItemTitle" data-win-bind="innerText: title"></h2>
    </div>
  </div>
</div>
```

Note that a template must *always* be declared in markup before any controls that reference them: `WinJS.UI.processAll` must instantiate the template first because the collection control will be asking the template to render its contents for each item in the data source. Also remember from Chapter 4 that instantiating a template removes its contents from the DOM so that it cannot be altered at run time. You can see this when running the sample: expand the nodes in Visual Studio’s DOM Explorer or Blend’s Live DOM pane, and you’ll see the root `div` of the template but none of its children.

In the sample, the prosaically named `ItemTemplate` is made of an `img` element and another `div` containing an `h2`. The `overlay` class on that latter `div`, if you look at the Figure 5-1 carefully, is clearly styled with a partially transparent background color (see `css/default.css` for the `.overlaidItemTemplate .overlay` selector). This shows that you can use any elements you want in a template, including other WinJS controls. In the latter case, these are picked up when `WinJS.UI.process/processAll` is invoked on the template.³⁰

You can also see that the template uses WinJS data-binding attributes, where the `img.src`, `img.alt`, and `h2.innerText` properties are bound to data properties called `picture` and `title`. This shows how properties of two target elements can be bound to the same source property. (Remember that if you’re binding to properties of the WinJS control itself, rather than its child elements, those properties must begin with `winControl`.)

For the data source, the FlipView’s `itemDataSource` option is assigned the value of `DefaultData.bindingList.dataSource` that you can find in `js/DefaultData.js`:

```
var array = [
  { type: "item", title: "Cliff", picture: "images/Cliff.jpg" },
  { type: "item", title: "Grapes", picture: "images/Grapes.jpg" },
  { type: "item", title: "Rainier", picture: "images/Rainier.jpg" },
  { type: "item", title: "Sunset", picture: "images/Sunset.jpg" },
  { type: "item", title: "Valley", picture: "images/Valley.jpg" }
];
var bindingList = new WinJS.Binding.List(array);
```

³⁰ Note that for such controls to be fully interactive, assign the `win-interactive` class to them, otherwise the surrounding control (and this applies to ListView as well) will swallow input events before they reach those controls.

```
WinJS.Namespace.define("DefaultData", {
    bindingList: bindingList,
    array: array
});
```

We briefly met `WinJS.Binding.List` at the end of Chapter 4; its purpose is to turn an in-memory array into an observable data source for one-way binding. The `WinJS.Binding.List` wrapper is also necessary because the `FlipView` and `ListView` controls cannot work directly against a simple array, even for one-time binding. They expect their data sources to provide the methods of the `WinJS.UI.IListDataSource` interface. The `dataSource` property of a `WinJS.Binding.List`, as in `bindingList.dataSource`, provides exactly this, and you'll always use this property in conjunction with `FlipView` and `ListView`. (It exists for no other purpose, in fact.) If you forget and attempt to just bind to the `WinJS.Binding.List` directly, you'll see an exception that says, "Object doesn't support property or method 'createListBinding'."

Suffice it to say that `WinJS.Binding.List` will become your intimate friend for in-memory data sources. Of course, you won't typically be using hard-coded data like the sample. You'll instead load array data from a file or obtain it from a web service, at which point `WinJS.Binding.List` makes it accessible to collection controls.

Do note that `WinJS.Binding.List` fully supports dynamic data. If you look at its [reference page](#) in the documentation, you'll see that it looks a whole lot like a JavaScript array, with a `length` property and the whole set of array methods from `concat` and `indexOf` to `push`, `pop`, and `unshift`. This is entirely intentional: no need to make you relearn the basics!

It's also important to note with `FlipView`, as well as `ListView`, that setting the `itemDataSource` property for the control automatically sets up one-way binding, so any changes to the list object or even the array on which it is built will trigger an automatic update in the bound control.

Quickstart #2a: The HTML ListView Essentials Sample

As I said before, the basic mechanisms for data sources and templates apply to the `ListView` control exactly as it does to `FlipView`, which we can now see in the [HTML ListView Essentials sample](#) (shown in Figure 5-2), specifically its first two scenarios of creating the control and responding to item events.

Because `ListView` can display multiple items at the same time, it needs one more piece in addition to the data source and the template to describe how those items visually relate to one another. This is the `ListView`'s `layout` property, which we see in the markup for Scenario 1 of this sample along with a few other behavioral options (`html/scenario1.html`):

```
<div id="listView" data-win-control="WinJS.UI.ListView"
    data-win-options="{ itemDataSource: myData.dataSource,
        itemTemplate: smallListItemIconTemplate, selectionMode: 'none',
        tapBehavior: 'none', swipeBehavior: 'none', layout: { type: WinJS.UI.GridLayout } }">
</div>
```

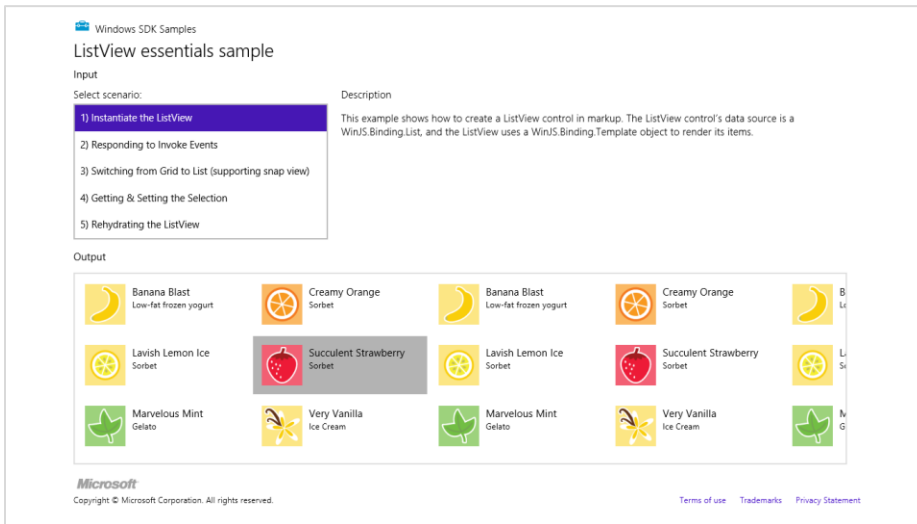


Figure 5-2 The HTML ListView Essentials sample.

The ListView's constructor, `WinJS.UI.ListView`, is, of course, called by the ubiquitous `WinJS.UI.processAll` when the page control is loaded. The data source for this list is set to `myData.dataSource` where `myData` is again a `WinJS.Binding.List` (defined at the top of `js/data.js` over a simple array) and its `dataSource` property provides the needed interface.

The control's item template is defined earlier in `default.html` with the id of `smallListIconTextTemplate` and is essentially the same sort of thing we saw with the FlipView (an `img` and some text elements), so I won't list it here.

In the control options we see three behavioral properties: `selectionMode`, `tapBehavior`, and `swipeBehavior`. These are all set to `'none'` in this sample to disable selection and click behaviors entirely, making the ListView a passive display. It can still be panned, but the items don't respond to input. (Also see the "Item Hover Styling" sidebar coming up.)

As for the `layout` property, this is an object of its own, whose `type` property indicates which layout to use. `WinJS.UI.GridLayout`, as we're using here, is a two-dimensional top-to-bottom then left-to-right algorithm, suitable for horizontal panning. WinJS provides another layout type called `WinJS.UI.ListLayout`, a one-dimensional top-to-bottom organization that's suitable for vertical panning, especially in snapped view. (We'll see this with the Grid App project template shortly; the ListView Essentials sample lacks a good snap view.)

Now while the ListView control in Scenario 1 only displays items, we often want those items to respond to a click or tap. Scenario 2 shows this, where the `tapBehavior` property is set to `'invoke'` (see `html/scenario2.html`). This is the same as using `tapBehavior: WinJS.UI.TapBehavior-toggleSelect`, as that's just defined in the [enumeration](#) as "invoke". This behavior will select or deselect and item, depending on its state, and then invoke it. Other variations are `directSelect`,

where an item is always selected and then invoked, and `invokeOnly` where the item is invoked without changing the selection state. You can also set the behavior to `none` so that clicks and taps are ignored.

When an item is invoked, the ListView control fires an `itemInvoked` event. You can wire up a handler by using either `addEventListener` or the ListView's `oniteminvoked` property. Here's how Scenario 2 does it (`js/scenario2.js`):

```
var listView = element.querySelector('#listView').winControl;

function itemInvokedHandler(eventObject) {
    eventObject.detail.itemPromise.done(function (invokedItem) {
        // Act on the item
    });
}

listView.addEventListener("iteminvoked", itemInvokedHandler, false);
```

Note that we're listening for the event on the WinJS *control*, but it also works to listen for the event on the containing element thanks to bubbling. This can be helpful if you need to add listeners to a control before it's instantiated, since the containing element will already be there in the DOM.

In the code above, you could also assign a handler by using the `listView.oniteminvoked` property directly, or you can specify the handler in the `iteminvoked` property `data-win-options`. In any case, the event object you receive in the handler contains a *promise* for the invoked item, not the item itself, so you need to call its `done` method to obtain the actual item data if needed. It's also good to know that you should never change the ListView's data source properties directly within an `iteminvoked` handler (because you'll probably cause an exception). If you have need to do that, place the change code inside `setImmediate` so that you can yield back to the UI thread first.

Sidebar: Item Hover Styling

While disabling selection and tap behaviors on a ListView creates a passive control, hovering over items with the mouse (or suitable touch hardware) still highlights each item; refer back to Figure 5-2. But you can control this using the `.win-container:hover` pseudo-selector for the desired control. For example, the following style rule removes the hover effect entirely:

```
#myListView .win-container:hover {
    background-color: transparent;
    outline: 0px;
}
```

Quickstart #2b: The ListView Grouping Sample

Displaying a list of items is great, but more often than not, a collection really needs another level of organization—what we call grouping. This is readily apparent when I open the file drawer next to my desk, which contains a collection of various important and not so important papers. Right away, on the file folder tabs, I see my groups: Taxes, Financials, Community, PGE, Insurance, Cars, Writing Projects, and Miscellany (among others). Clearly, then, we need a grouping facility within a collection control

and ListView is happy to oblige!

A core demonstration of grouping can be found in the [HTML ListView Grouping and Semantic Zoom sample](#) (shown in Figure 5-3). As with the Essentials sample, the code in `js/groupedData.js` contains a lengthy in-memory array around which we create a `WinJS.Binding.List`. Here's a condensation to show the item structure (I'd show the whole array, but this is making me hungry for some dessert!):

```
var myList = new WinJS.Binding.List([
  { title: "Banana Blast", text: "Low-fat frozen yogurt", picture: "images/60Banana.png" },
  { title: "Lavish Lemon Ice", text: "Sorbet", picture: "images/60Lemon.png" },
  { title: "Creamy Orange", text: "Sorbet", picture: "images/60Orange.png" },
  ...
]);
```

Here we have a bunch of items with `title`, `text`, and `picture` properties, and we can group them really any way we like (and even change the groupings on the fly). As Figure 5-3 shows, the sample groups these by the first letter of the title.

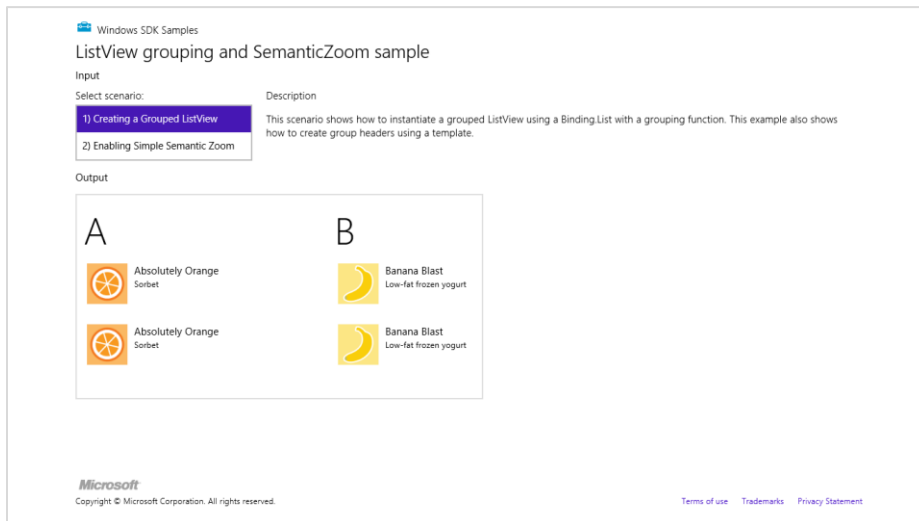


Figure 5-3 The HTML ListView Grouping and Semantic Zoom sample.

If you take a peek at the [ListView reference](#), you'll see that it works with two templates and two collections: that is, alongside its `itemTemplate` and `itemDataSource` properties are ones called `groupHeaderTemplate` and `groupDataSource`. These are used with the ListView's `GridLayout` (the default) to organize the groups and create the headers above the items.

The header template in `html/scenario1.html` is very simple (and the item template is like what we've already seen):

```
<div id="headerTemplate" data-win-control="WinJS.Binding.Template">
  <div class="simpleHeaderItem">
    <h1 data-win-bind="innerText: title"></h1>
```

```

    </div>
</div>

```

This is referenced in the control declaration (other options omitted):

```

<div id="listView" data-win-control="WinJS.UI.ListView"
    data-win-options="{ groupDataSource: myGroupedList.groups.dataSource,
                        groupHeaderTemplate: headerTemplate }">
</div>

```

For the data sources, you can see that we're now using a variable called `myGroupedList` with a property inside it called `groups`. What's all this about?

Well, let's take a short conceptual detour. Although computers have no problem chewing on a bunch of raw data like the `myList` array, human beings like to view data with a little more organization. The three primary ways of doing this are *grouping*, *sorting*, and *filtering*. Grouping organizes items into groups, as shown in Figure 5-3; sorting orders items according to various rules; and filtering provides a subset of items that match certain criteria. In all three cases, however, you don't want such operations to actually change the underlying data: a user might want to group, sort, or filter the same data in different ways from moment to moment.

Grouping, sorting, and filtering, then, are thus referred to as *projections* of the data: they're all connected to the same underlying data such that a change to an item in the projection will be propagated back to the source, just as changes in the source are reflected in the projection.

The `WinJS.Binding.List` object provides methods to create these projections: `createGrouped`, `createSorted`, and `createFiltered`. Each method produces a special form of a `WinJS.Binding.List`: `GroupedSortedListProjection`, `SortedListProjection`, and `FilteredListProjection`, respectively. That is, each projection is a bindable list in itself, with a few extra methods and properties that are specific to the projection. You can even create a projection from a projection. For instance, `createGrouped(...).createFiltered(...)` will create a filtered projection on top of a grouped projection. (Note, however, that the list's `sort` method applies the sorting in-place, just like the JavaScript array's `sort`.)

Now that we know about projections, we can see how `myGroupedList` is created:

```

var myGroupedList = myList.createGrouped(getGroupKey, getGroupData, compareGroups);

```

This method takes three functions. The first associates an item with a group: it receives an item and returns the appropriate group string, known as the key. The key—which must be a string—can be something that's directly included in an item or it can be derived from item properties. In the sample, the `getGroupKey` function returns the first character of the item's `title` property (in upper case):

```

function getGroupKey(dataItem) {
    return dataItem.title.toUpperCase().charAt(0);
}

```

Be clear that this first function, referred to as the *group key* function, determines *only* the association between the item and a group, nothing more. It also gets called for every item in the collection when

`createGrouped` is called, so it should be a quick operation.³¹ The data for the groups themselves, which is the collection to which the header template is bound to, isn't actually created until the group projection's `groups` method is invoked, as happens when our `ListView`'s `groupedDataSource` option gets processed. At that point, the second function passed to `createGrouped`—the *group data* function—gets called only once per group with a *representative* item for that group. In response, your function returns an object for that group containing whatever properties you need for data binding.

In the sample, the `getGroupData` function (passed to `createGrouped`) simply returns an object with a single `title` property that's the same as the group key, but of course you can make that value anything you want:

```
function getGroupData(dataItem) {
    return {
        title: dataItem.title.toUpperCase().charAt(0)
    };
}
```

If I could rewrite this sample, I'd probably name the `title` property of this group data object something more distinct, like *groupTitle*, so it'd be very clear that it has nothing whatsoever to do with the same-named property of the *items*. I point this out because if you look in `scenario1.html` at the header template and the item template, you'll see `title` properties used in both. Yet the data context is entirely different in both cases: for the header template, it's the collection generated by the return values of your group data function; for the item template, it's the grouped projection from `WinJS.Binding.List.createGrouped`. Two different collections—remember that!

So why do we have the group data function separated out at all? Why not just create that collection automatically from the group keys? It's because you often want to include additional properties within the group data that you might want to use in the header template or in a zoomed-out view (with semantic zoom). Think of your group data function as providing a kind of summary information for each group. (The header text is really only the most basic such summary.) Since this function is only called once per group, rather than once per item, it's the proper time to calculate or otherwise retrieve summary-level data. For example, to show an item count in the group headers, we just need to include that property in the objects returned by the group data function, then data-bind an element in the header template to that property.

In the sample, we can use `WinJS.Binding.List.createFiltered` to obtain a projection of the list filtered by the current key. The `length` property of this projection is then the number of items in the group (note that I'm now reusing the group key function instead of repeating the code):

```
function getGroupData(dataItem) {
    var key = getGroupKey(dataItem);

    //Obtain a filtered projection of our list, checking for matching keys
    var filteredList = myList.createFiltered(function (item) {
```

³¹ If deriving the group key from an item at run time required an involved process, you'll improve overall performance by storing a prederived key in the item instead.

```

    return key == getGroupKey(item);
  });

  return {
    title: key,
    count: filteredList.length
  };
}

```

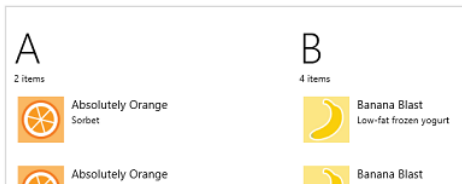
With this `count` property in the collection, we can use it in the header template:

```

<div id="headerTemplate" data-win-control="WinJS.Binding.Template" style="display: none">
  <div class="simpleHeaderItem">
    <h1 data-win-bind="innerText: title"></h1>
    <h6><span data-win-bind="innerText: count"></span> items</h6>
  </div>
</div>

```

After a small tweak in `css/scenario1.css`—changing the `simpleHeaderItem` class height to 65px to make a little more room—the list will now appears as follows:



Finally, back to `WinJS.Binding.List.createGrouped`, the third (and optional) function here is a *group sorter* function, which is called to sort the group data collection and therefore the order in which those groups appear in the `ListView`.³² This function receives two group keys and returns zero if they're equal, a negative number if the first key sorts before the second, and a positive if the second sorts before the first. The `compareGroups` function in the sample does an alphabetical sort:

```

function compareGroups(left, right) {
  return left.toUpperCase().charCodeAt(0) - right.toUpperCase().charCodeAt(0);
}

```

For a two-level sort, first by the descending item count and then by the first character, we could write the following:

```

function compareGroups(left, right) {
  var leftLen = filteredLengthFromKey(left);
  var rightLen = filteredLengthFromKey(right);

  if (leftLen != rightLen) {
    return rightLen - leftLen;
  }
}

```

³² This is entirely separate from creating a sorted projection of the *items*, for which you'd use `WinJS.Binding.List.createSorted`.

```

    return left.toUpperCase().charCodeAt(0) - right.toUpperCase().charCodeAt(0);
}

function filteredLengthFromKey(key) {
    var filteredList = myData.list.createFiltered(function (item) {
        return key == getGroupKey(item);
    });

    return filteredList.length;
}

```

Debugging Your Grouping Functions

If your various grouping functions don't seem to be working right, you can set breakpoints and step through the code a few times, but this becomes tedious as the functions are called many, many times for even modest collections. Instead, try using `console.log` to emit the parameters sent to those functions and/or your return values. To see what's coming into the group sorting function, for example, try this code:

```
console.log("Comparing left = " + left + " to right = " + right);
```

ListView in the Grid App Project Template

Now that we've covered the details of the ListView control and in-memory data sources, we can finally understand the rest of the Grid App project template in Visual Studio and Blend. As we covered in Chapter 3 (in the "The Navigation Process and Navigation Styles" section), this project template provides an app structure built around page navigation: the home page (pages/groupedItems) displays a collection of sample data (see js/data.js) in a ListView control, where each item's presentation is described by a [WinJS.Binding.Template](#) as are the group headings. Figure 5-4 shows the layout of the home page and identifies the relevant ListView elements. As we also discussed before, tapping an item navigates to the pages/itemDetail page and tapping a heading navigates to the pages/groupDetail page, and now we can see how that all works with the ListView control.

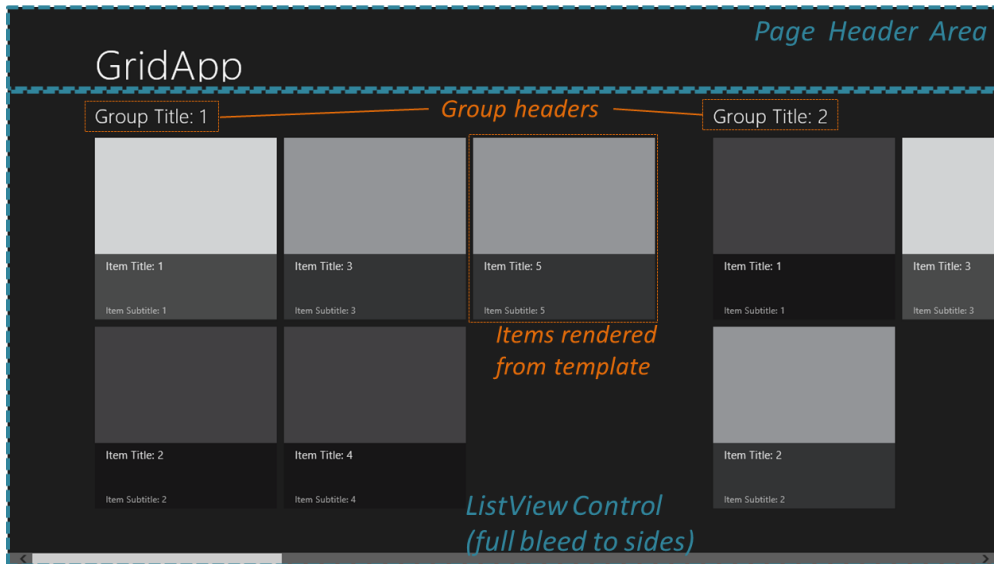


Figure 5-4 ListView elements as shown in the Grid App template home page. (All colored items are added labels and lines.)

The ListView, as in Figure 5-4, occupies the lower portion of the app's contents. Because it can pan horizontally, it actually extends all the way across; various CSS margins are used to align the first items with the layout silhouette while allowing them to bleed to the left when the ListView is panned.

There's quite a bit going on with the ListView in this project, so we'll take one part at a time. For starters, the control's markup in `groupedItems.html` is very basic, where the only option is to indicate that the items have no selection behavior:

```
<div class="groupeditemslist win-selectionstylefilled" aria-label="List of groups"
    data-win-control="WinJS.UI.ListView" data-win-options="{ selectionMode: 'none' }">
</div>
```

Switching over to `groupedItems.js`, the page's `ready` method handles initialization:

```
ready: function (element, options) {
    var listView = element.querySelector(".groupeditemslist").winControl;
    listView.groupHeaderTemplate = element.querySelector(".headerTemplate");
    listView.itemTemplate = element.querySelector(".itemtemplate");
    listView.oniteminvoked = this._itemInvoked.bind(this);

    // (Keyboard handler initialization omitted)...

    this.initializeLayout(listView, appView.value);
    listView.element.focus();
},
```

Here you can see that the control's templates can be set in code just as easily as from markup, and in this case we're using a class to locate the template element instead of an Id. Why does this work? It's

because we've actually been referring to elements the whole time: the app host automatically creates a variable for an element that's named the same as its Id. It's the same thing. In code you can also provide a function instead of a template, which allows you to dynamically render each item individually. More on this later.

You can also see how this page assigns a handler to the `itemInvoked` events (above `ready`), calling `WinJS.Navigation.navigate` to go to the `groupDetail` or `itemDetail` pages as we saw in Chapter 3:

```
_itemInvoked: function (args) {
    if (appView.value === appViewState.snapped) {
        // If the page is snapped, the user invoked a group.
        var group = Data.groups.getAt(args.detail.itemIndex);
        this.navigateToGroup(group.key);
    } else {
        // If the page is not snapped, the user invoked an item.
        var item = Data.items.getAt(args.detail.itemIndex);
        nav.navigate("/pages/itemDetail/itemDetail.html", {
            item: Data.getItemReference(item) });
    }
}

navigateToGroup: function (key) {
    nav.navigate("/pages/groupDetail/groupDetail.html", { groupKey: key });
},
```

In this case we retrieve item data from the underlying collection (the `getAt` methods) rather than using the item data itself, since group information needed for the first case isn't part of an item directly. We also see here that the page interprets item invocations differently depending on the view state. This is because it actually switches both its layout and its data source when the view state changes, as handled in the page's internal `_initializeLayout` method, called both on startup and from the page's `updateLayout` function:

```
initializeLayout: function (listView, viewState) {
    if (viewState === appViewState.snapped) {
        listView.itemDataSource = Data.groups.dataSource;
        listView.groupDataSource = null;
        listView.layout = new ui.ListLayout();
    } else {
        listView.itemDataSource = Data.items.dataSource;
        listView.groupDataSource = Data.groups.dataSource;
        listView.layout = new ui.GridLayout({ groupHeaderPosition: "top" });
    }
},
```

A `ListView`'s layout, in short, can be changed at any time by setting its `layout` property. When the view state is snapped, this is set to `WinJS.UI.ListLayout`, otherwise `WinJS.UI.GridLayout` (whose `groupHeaderPosition` property can be `"top"` or `"left"`). You can also see that you can change a `ListView`'s data source on the fly: in snapped state it's a list of groups, otherwise it's the list of items.

I hope you can now see why I introduced page navigation well before we got to `ListView`, because this project gets quite complicated down in its depths! In any case, let's now look at the templates for

this page (groupedItems.html):

```
<div class="headertemplate" data-win-control="WinJS.Binding.Template">
  <button class="group-header win-type-x-large win-type-interactive"
    data-win-bind="groupKey: key" role="link" tabindex="-1" type="button"
    onclick="Application.navigator.pageControl.navigateToGroup(event.srcElement.groupKey)" >
    <span class="group-title win-type-ellipsis" data-win-bind="textContent: title"></span>
    <span class="group-chevron"></span>
  </button>
</div>

<div class="itemtemplate" data-win-control="WinJS.Binding.Template">
  <div class="item">
    
    <div class="item-overlay">
      <h4 class="item-title" data-win-bind="textContent: title"></h4>
      <h6 class="item-subtitle win-type-ellipsis"
        data-win-bind="textContent: subtitle"></h6>
    </div>
  </div>
</div>
</div>
```

Again, we have the same use of `WinJS.Binding.Template` and various bits of data-binding syntax sprinkled around the markup, not to mention the `click` handler assigned to the header text itself which, like an item in snap view, navigated to the group detail page.

As for the data itself (that you'll likely replace), this is defined in `js/data.js` again as an in-memory array that feeds into `WinJS.Binding.List`. In the `sampleItems` array each item is populated with inline data or other variable values. Each item also has a `group` property that comes from the `sampleGroups` array. Unfortunately, this latter array has almost identical properties as the items array, which can get confusing. To help clarify that a bit, here's the complete property structure of an item:

```
{
  group : {
    key,
    title,
    subtitle,
    backgroundImage,
    description
  },
  title,
  subtitle,
  description,
  content,
  backgroundImage
}
```

As we saw with the `ListView` grouping sample earlier, the `Grid App` project template uses `createGrouped` to set up the data source. What's interesting to see here is that it sets up an initially empty list, creates the grouped projection (omitting the optional sorter function), and then adds the items by using the list's `push` method:


```

var list = new WinJS.Binding.List();
var groupedItems = list.createGrouped(
    function groupKeySelector(item) { return item.group.key; },
    function groupDataSelector(item) { return item.group; }
);

generateSampleData().forEach(function (item) {
    list.push(item);
});

```

This clearly shows the dynamic nature of lists and ListView: you can add and remove items from the data source, and one-way binding will make sure the ListView is updated accordingly. In such cases you do *not* need to refresh the ListView's layout—that happens automatically. I say this because there's been some confusion with the ListView's `forceLayout` method, which you only need to call, as the documentation states, "when making the ListView visible again after its `style.display` property had been set to 'none'." You'll find, in fact, that the Grid App code doesn't use this method at all.

In `data.js` there are also a number of other utility functions, such as `getItemsFromGroup`, which uses `WinJS.Binding.List.createFiltered` as we did earlier. Other functions provide for cross-referencing between groups and items, as it's needed to navigate between the items list, group details (where that page shows only items in that group), and item details. All of these functions are wrapped up in a namespace called `Data` at the bottom of `data.js`, so references to anything from this file are prefixed elsewhere with `Data..`

And with that, I think you'll be able to understand everything that's going on in the Grid App project template and you'll be able to adapt it to your own needs. Just remember that all the sample data is intended to be wholly replaced with real data that you obtain from other sources, like a file or `WinJS.xhr`, and that you can wrap with `WinJS.Binding.List`. Some further guidance on this can be found in the [Create a blog reader tutorial](#) on the Windows Dev Center, and although the tutorial uses the Split App project template, there's enough in common with the Grid App project template that the discussion is really applicable to both.

The Semantic Zoom Control

Since we've already loaded up the [HTML ListView Grouping and Semantic Zoom sample](#), and have completed our first look at the collection controls, now is a good time to check out another very interesting WinJS control: [Semantic Zoom](#).

Semantic zoom lets users easily switch between two views of the same data: a zoomed-in view that provides details and a zoomed-out view that provides more summary-level information. The primary use case for semantic zoom is a long list of items (especially ungrouped items), where a user will likely get really bored of panning all the way from one end to the other, no matter how fun it is to swipe the screen with a finger. With semantic zoom, you can zoom out to see headers, categories, or some other condensation of the data, and then tap on one of those items to zoom back into its section or group. The design guidance recommends having the zoomed-out view fit on one to three screenfuls at most,

making it very easy to see and comprehend the whole data set.

Go ahead and try semantic zoom through Scenario 2 of the ListView Grouping and Semantic Zoom sample. To switch between the views, use pinch-zoom touch gestures, Ctrl+/Ctrl- keystrokes, Ctrl+mouse wheel, and/or a small zoom button that automatically appears in the lower-right corner of the control, as shown in Figure 5-5. When you zoom out, you'll see a display of the group headers, as also shown in the figure.

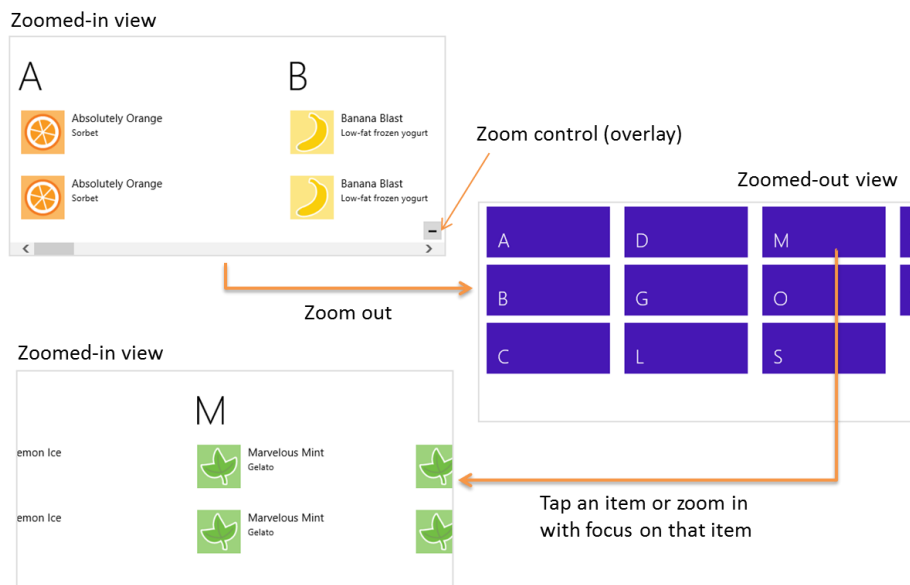


Figure 5-5 Semantic zoom between the two views in the ListView Grouping and Semantic Zoom sample.

The control itself is quite straightforward to use. In markup, declare a WinJS control using the `WinJS.UI.SemanticZoom` constructor. Within that element you then declare two (and only two) child elements: the first defining the zoomed-in view, and the second defining the zoomed-out view. Here's how the sample does it with two ListView controls (plus the template used for the zoomed-out view):

```
<div id="semanticZoomTemplate" data-win-control="WinJS.Binding.Template" >
  <div class="semanticZoomItem">
    <h2 class="semanticZoomItem-Text" data-win-bind="innerText: title"></h2>
  </div>
</div>

<div id="semanticZoomDiv" data-win-control="WinJS.UI.SemanticZoom">
  <div id="zoomedInListView" data-win-control="WinJS.UI.ListView"
    data-win-options="{ itemDataSource: myGroupedList.dataSource,
      itemTemplate: mediumListIconTextTemplate,
      groupDataSource: myGroupedList.groups.dataSource,
      groupHeaderTemplate: headerTemplate,
      selectionMode: 'none', tapBehavior: 'none', swipeBehavior: 'none' }">
  </div>
</div>
```

```

<div id="zoomedOutListView" data-win-control="WinJS.UI.ListView"
    data-win-options="{ itemDataSource: myGroupedList.groups.dataSource,
        itemTemplate: semanticZoomTemplate,
        selectionMode: 'none', tapBehavior: 'invoke', swipeBehavior: 'none' }" >
    </div>
</div>

```

The first child, *zoomedInListView*, is just like the *ListView* for Scenario 1 with group headers and items; the second, *zoomedOutListView*, uses the groups as items and renders them with a different template. The semantic zoom control simply switches between the two views on the appropriate input gestures. When the zoom changes, the semantic zoom control fires a `zoomchanged` event where the `args.detail` value in the handler is `true` when zoomed out, `false` when zoomed in. You might use this event to make certain app bar commands available for the different views, such as commands in the zoomed-out view to change sorting or filtering, which would then affect how the zoomed-in view is displayed.

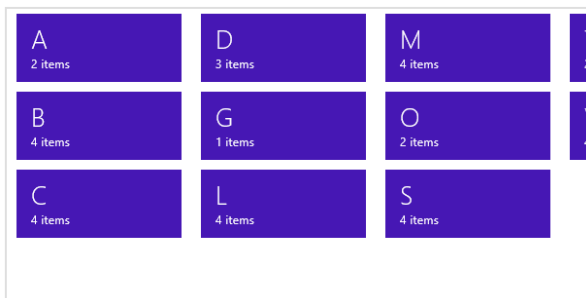
The control has a few other properties, such as `enableButton` (a Boolean to control the visibility of the overlay button; default is `true`), `locked` (a Boolean that disables zooming in either direction and can be set dynamically to lock the current zoom state; default is `false`), and `zoomedOut` (a Boolean indicating if the control is zoomed out, so you can initialize it this way; default is `false`). There is also a `forceLayout` method that's used in the same case as the *ListView*'s `forceLayout`: namely, when you remove a `display: none` style.

The `zoomFactor` property is an interesting one that determines how the control animates between the two views. The animation is a combination of scaling and cross-fading that makes the zoomed-out view appear to drop down from or rise above the plane of the control, depending on the direction of the switch, while the zoomed-in view appears to sink below or come up to that plane. To be specific, the zoomed-in view scales between 1 and `zoomFactor` while transparency goes between 1 and 0, and the zoomed-out view scales between $1/\text{zoomFactor}$ and 1 while transparency goes between 0 and 1. The default value for `zoomFactor` is 0.65, which creates a moderate effect. Lower values (minimum is 0.2) emphasize the effect, and higher values (maximum is 0.8) minimize it.

Where styling is concerned, you do most of what you need directly to the Semantic Zoom's children. However, to style the Semantic Zoom control itself you can override styles in `win-semanticzoom` (for the whole control) and `win-semanticzoomactive` (for the active view). The `win-semanticzoombutton` style also lets you style the zoom control button if needed.

It's important to understand that semantic zoom is intended to switch between two views of the same data and not to switch between completely different data sets (see [Guidelines and checklist for the Semantic Zoom control](#)). Also, the control does not support nesting (that is, zooming out multiple times to different levels). Yet this doesn't mean you have to use the same kind of control for both views: the zoomed-in view might be a list, and the zoomed-out view could be a chart, a calendar, or any other visualization that makes sense. The zoomed-out view, in other words, is a great place to show summary data that would be otherwise difficult to derive from the zoomed-in view. For example, using the same changes we made to include the item count with the group data for Scenario 1 (see

“Quickstart #2b” above), we can just add a little more to the zoomed-out item template:



The other thing you need to know is that the semantic zoom control does not work with arbitrary child elements. An exception about a missing `zoomableView` property will tell you this! Each child control must provide an implementation of the [WinJS.UI.IZoomableView interface](#) through a property called `zoomableView`. Of all built-in HTML and WinJS controls, only `ListView` does this, which is why you typically see semantic zoom in that context, and why you can't nest semantic zooms! However, you can certainly provide this interface on a custom control, where the object returned by the constructor should contain a `zoomableView` property, which is an object containing the methods of the interface. Among these methods are `beginZoom` and `endZoom` for obvious purposes, and `getCurrentItem` and `setCurrentItem` that enable the semantic zoom control to zoom in to the right group when it's tapped in the zoomed-out view.

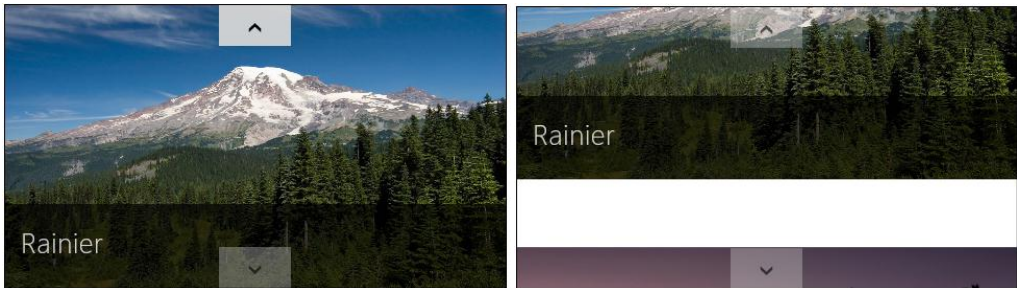
For more details, check out the [HTML SemanticZoom for custom controls sample](#) in the Windows SDK, which also serves as another example of a custom control.

FlipView Features and Styling

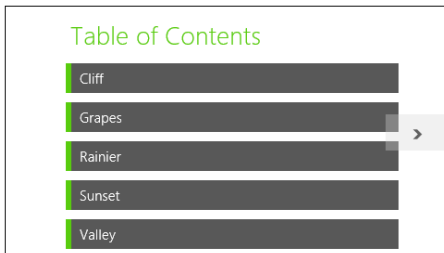
For all the glory that `ListView` merits as the richest and most sophisticated control in all of WinJS, we don't want to forget the humble `FlipView`! Thus before we delve wholly into `ListView`, let's spend a few pages covering `FlipView` and its features through the other scenarios in the [FlipView Control sample](#). It's worth mentioning too that although this sample demonstrates the control's capabilities in a relatively small area, a `FlipView` can be any size, even occupying most of the screen. A common use for the control, in fact, is to let users flip through full-sized images in a photo gallery. Of course, the control can be used anywhere it's appropriate, large or small. See [Guidelines for FlipView controls](#) for more on how best to use the control.

Anyway, Scenario 2 in the sample (“Orientation and Item Spacing”) demonstrates the control's `orientation` property. This determines the placement of the arrow controls: left and right (`horizontal`) or top and bottom (`vertical`) as shown below. It also determines the enter and exit animations of the items and whether the control uses the left/right or up/down arrow keys for keyboard navigation. This scenario also let you set the `itemSpacing` property, which determines the amount of space between items when you swipe items using touch (below right). Its effect is not visible

when using the keyboard or mouse to flip; to see it, you may need to use touch emulation in the Visual Studio simulator to partly drag between items.

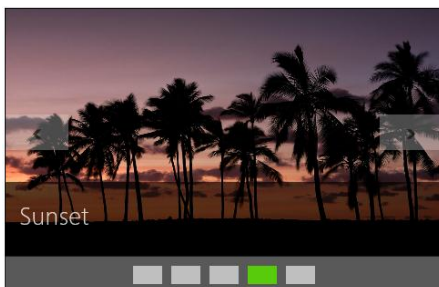


Scenario 3 (“Using interactive content”) shows the use of a *template function* instead of a declarative template. We’ll talk more of such functions in “How Templates Really Work” later in this chapter, but put simply, a template function or *renderer* creates elements and sets their properties procedurally, which is essentially what [WinJS.Binding.Template](#) does from the markup you give it. This allows you to render an item differently (that is, create different elements) depending on its actual data. In Scenario 3, the data source contains a “table of contents” item at the beginning, for which the renderer (a function called `mytemplate` in `interactiveContent.js`) creates a completely different item:



The scenario also sets up a listener for `click` events on TOC entries, the handler for which flips to the appropriate item by setting the FlipView’s `currentPage` property. The picture items then have a back link to the TOC. See the `clickHandler` function in the code for both of these actions.

Scenario 4 (“Creating a context control”) demonstrates adding a navigation control overlay to each item:



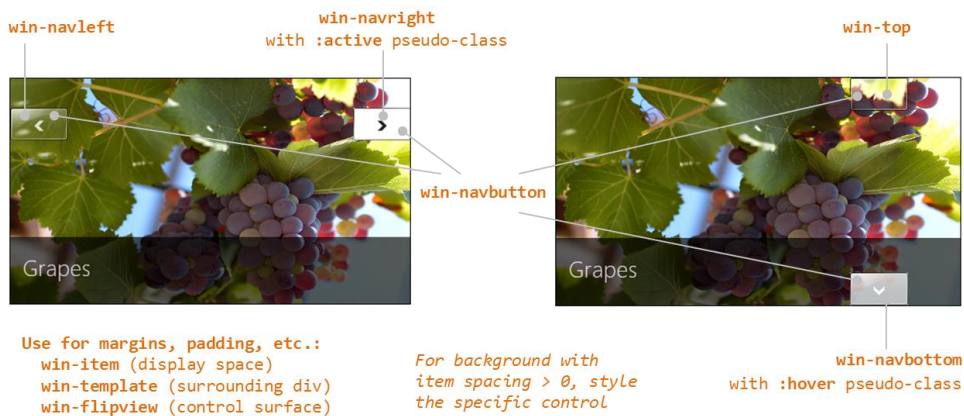
The items themselves are again rendered using a declarative template, which in this case just contains a placeholder `div` called `ContextContainer` for the navigation control (contextControl.html):

```
<div>
  <div id="contextControl_FlipView" class="flipView" data-win-control="WinJS.UI.FlipView"
    data-win-options="{ itemDataSource: DefaultData.bindingList.dataSource,
      itemTemplate: contextControl_ItemTemplate }">
    </div>
    <div id="ContextContainer"></div>
  </div>
```

When the control is initialized in the `processed` method of `contextControl.js`, the sample calls the `FlipView`'s async `count` method. The completed handler, `countRetrieved`, then creates the navigation control using a row of styled radiobuttons in a row. The `onpropertychange` handler for each radiobutton then sets the `FlipView`'s `currentPage` property.

Scenario 4 also sets up listeners for the `FlipView`'s `pageselected` and `pagevisibilitychanged` events. The first is used to update the navigation radiobuttons when the user flips between pages. The other is used to prevent clicks on the navigation control while a flip is happening. (The event occurs when an item changes visibility and is fired twice for each flip, once for the previous item, and again for the new one.)

Scenario 5 ("Styling Navigation Buttons") demonstrates the styling features of the `FlipView`, which involves various `win-*` styles and pseudo-classes as shown here:



If you were to provide your own navigation buttons in the template (wired to the `next` and `previous` methods), hide the default by adding `display: none` to the `<control selector>.win-navbutton` style rule.

Finally, there are a few other methods and events for the `FlipView` that aren't used in the sample, so here's a quick rundown of those:

- `pageCompleted` is an event that is raised when a flip to a new item is fully completed (that is, the new item has been rendered). In contrast, the aforementioned

`pageselected` will fire when a *placeholder* item (not fully rendered) has been animated in. See “Template Functions (Part 2)” at the end of this chapter.

- `datasourcecountchanged` is an event raised for obvious purpose, which something like Scenario 4 would use to refresh the navigation control if items could be added or removed from the data source.
- `next` and `previous` are methods to flip between items (like `currentPage`), which would be useful if you provided your own navigation buttons.
- `forceLayout` is a method to call specifically when you make a FlipView visible by removing a `display: none` style. (The FlipView sample actually calls this whenever you change scenarios, but it’s not necessary because it never changes the style.)
- `setCustomAnimations` allows you to control the animations used when flipping forward, flipping backward, and jumping to a random item.

For details on all of these, refer to the [WinJS.UI.FlipView object](#) documentation.

Data Sources

In all the examples we’ve seen thus far, we’ve been using an in-memory data source built on `WinJS.Binding.List`. Clearly, however, there are other types of data sources and it certainly doesn’t make sense to load everything into memory first. How, then, do we work with such sources?

WinJS provides some help in this area. First is the `WinJS.UI.StorageDataSource` object that works with files in the file system, as the next section demonstrates with a FlipView and the Pictures Library. The other is `WinJS.UI.VirtualizedDataSource`, which is meant for you to use as a base class for a custom data source of your own, an advanced scenario that we’ll touch on only briefly.

A FlipView Using the Pictures Library

For everything we’ve seen in the FlipView sample already, it really begs for the ability to do something completely obvious: flip through pictures files in a folder. Using what we’ve learned so far, how would we implement something like that? We already have an item template containing an `img` tag, so perhaps we just need some URIs for those files. Perhaps we could make an array of these using an API like `Windows.Storage.KnownFolders.picturesLibrary.GetFilesAsync` (declaring the pictures library capability in the manifest, of course!), which would give us a bunch of `StorageFile` objects for which we could call `URL.createObjectURL`. We could store those URIs in an array and then wrap it up with `WinJS.Binding.List`:

```
var myFlipView = document.getElementById("pictures_FlipView").winControl;  
  
Windows.Storage.KnownFolders.picturesLibrary.GetFilesAsync()  
    .done(function (files) {
```

```

var pixURLs = [];

files.forEach(function (item) {
    var url = URL.createObjectURL(item, {oneTimeOnly: true });

    pixURLs.push({type: "item", title: item.name, picture: url });
});

var pixList = new WinJS.Binding.List(pixURLs);
myFlipView.itemDataSource = pixList.dataSource;
});

```

Although this approach works, it can start to consume quite a bit of memory with a larger number of high-resolution pictures because each picture has to be fully loaded to be displayed in the FlipView. This might be just fine for your scenario but in other cases would consume more resources than necessary. It also has the drawback that the images are just stretched or compressed to fit into the FlipView without any concern for aspect ratio, and this doesn't produce the best results.

A better approach is to use the [WinJS.UI.StorageDataSource](#) that again works directly with the file system instead of an in-memory array. I've implemented this as a Scenario 8 in the modified FlipView sample code in this chapter's companion content. (Another example can be found in the [StorageDataSource and GetVirtualizedFilesVector sample](#).) Here we can use a shortcut to get a data source for the Pictures library:

```
myFlipView.itemDataSource = new WinJS.UI.StorageDataSource("Pictures");
```

"Pictures" is a shortcut because the first argument to [StorageDataSource](#) is actually something called a file query that comes from the [Windows.Storage.Search](#) API, a subject we'll see in more detail in Chapter 8, "State, Settings, Files, and Documents." These queries, which feed into the powerful [Windows.Storage.StorageFolder.createFileQueryWithOptions](#) function, are ways to enumerate files in a folder along with metadata like album covers, track details, and thumbnails that are cropped to maintain the aspect ratio. Shortcuts like "Pictures" (also "Music", "Documents", and "Videos" that all require the associated capability in the manifest) just create typical queries for those document libraries.

The caveat with [StorageDataSource](#) is that it doesn't directly support one-way binding, so you'll get an exception if you try to refer to item properties directly in a template. To work around this, you have to explicitly use [WinJS.Binding.oneTime](#) as the initializer function for each property:

```

<div id="pictures_ItemTemplate" data-win-control="WinJS.Binding.Template">
    <div class="overlaidItemTemplate">
        <img class="image" data-win-bind="src: thumbnail InitFunctions.thumbURL;
            alt: name WinJS.Binding.oneTime" />
        <div class="overlay">
            <h2 class="ItemTitle" data-win-bind="innerText: name WinJS.Binding.oneTime"></h2>
        </div>
    </div>
</div>

```

In the case of the `img.src` property, the file query gives us items of type [Windows.Storage-](#)

[BulkAccess.FileInformation](#) (the `source` variable in the code below), which contains a thumbnail image, not a URL. To convert that image data into a URL, we need to use our own binding initializer:

```
WinJS.Namespace.define("InitFunctions", {
    thumbURL: WinJS.Binding.initializer(function (source, sourceProp, dest, destProp) {
        if (source.thumbnail) {
            dest.src = URL.createObjectURL(source.thumbnail, { oneTimeOnly: true });
        }
    })
});
```

In this initializer, the `src : thumbnail` part of `data-win-bind` is actually ignored because we're just setting the img's `src` property directly to `source.thumbnail`; this is just a form of one-way binding.

Note that thumbnails aren't always immediately available in the `FileInformation` object, which is why we have to verify that we actually have one before creating a URL for it. This means that quickly flipping through the images might show some blanks. To solve this particular issue, we can listen for the `FileInformation.onthumbnailupdated` event and update the item at that time. The best way to accomplish this is to use the [StorageDataSource.loadThumbnail](#) helper, which helps avoid subtle memory leak problems in this particular process. You can use this method within a binding initializer, as demonstrated in Scenario 1 of the aforementioned [StorageDataSource and GetVirtualizedFilesVector sample](#), or within a rendering function that takes the place of the declarative template. We'll do this for our FlipView sample later on, in "How Templates Really Work," which also lets us avoid the one-time binding tricks.

As a final note, Scenario 6 of the FlipView sample contains another example of a different data source, specifically one working with Bing Search. For that, let's look at custom data sources.

Custom Data Sources

Now that we've seen a collection control like FlipView working against two different data sources, you're probably starting to correctly guess that all data sources share some common characteristics and a common programmatic interface. This is demonstrated again in Scenario 6 of the FlipView sample as well as in the [HTML ListView working with data sources sample](#) shown in Figure 5-6, as we'll see in this section.

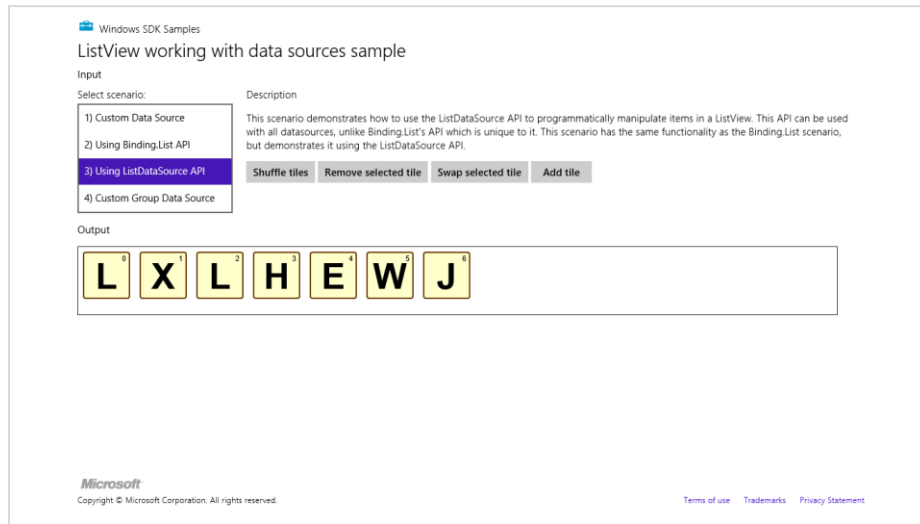


Figure 5-6 The SDK's Working with Data Sources sample.

Scenarios 2 and 3 of this sample both work against a [WinJS.Binding.List](#) data source, as we've already seen, and provide buttons to manipulate that data source. Those changes are reflected in the output. The difference between the two scenarios is that Scenario 2 manipulates the data through [WinJS.Binding.List](#) methods like [move](#), whereas Scenario 3 manipulates the underlying data source directly through a more generic [ListDataSource API](#).

Because of data binding, changes to the data are reflected in the ListView control either way, but there are three important differences. First, the [ListDataSource](#) interface is common to all data sources, so any code you write against it will work for any kind of data source. Second, its methods are generally asynchronous because a data source might be connected to an online service or other such resource. Third, [ListDataSource](#) provides for batching changes together by calling [beginEdits](#), which will defer any change notifications to any external bound objects until [endEdits](#) is called. This allows you to do bulk data editing in ways that can improve ListView performance.

Scenarios 1 and 4 of the sample then demonstrate how to create custom data sources. Scenario 1 creates a data source for Bing searches; Scenario 4 creates one for an in-memory array that you could adapt to work against a data feed that's only brought down from a service a little at a time. What's important for all these is that they implement something called a data adapter, which is an object with the methods of the [WinJS.UI.IListDataAdapter interface](#). This provides for capabilities like caching, virtualization, change detection, and so forth. Fortunately, you get most of these methods by deriving your class from [WinJS.UI.VirtualizedDataSource](#) and then implementing those methods you need to customize. In the sample, for instance, the [bingImageSearchDataSource](#) is defined as follows (see [js/BingImageSearchDataSource.js](#)):

```
bingImageSearchDataSource = WinJS.Class.derive(WinJS.UI.VirtualizedDataSource,
    function (devkey, query) {
        this._baseDataSourceConstructor(new bingImageSearchDataAdapter(devkey, query));
    });
```

```
});
```

where the `BingImageSearchDataAdapter` class implements only the `getCount` and `itemsFromIndex` methods directly.

For a further deep-dive on this subject beyond the sample, I refer you to a session from the 2011 //Build conference entitled [APP210-T: Build data-driven collection and list apps in HTML5](#). Some of the details have since changed (like the `ArrayDataSource` is now `WinJS.Binding.List`), but on the whole it very much explains all the mechanisms. It's also helpful to remember that you can use other languages to write custom data sources as well, languages that could offer much higher performance within the data source—or have access to higher-performant APIs—than JavaScript. That subject is well beyond the scope of this book, but I at least wanted to mention the possibility.

How Templates Really Work

Earlier, when we looked at the Grid App project template, I mentioned that you can use a function instead of a declarative template for properties like `itemTemplate` (FlipView and ListView) and `groupHeaderTemplate` (ListView). This is a very important capability because it allows you to dynamically render each item in a collection individually, using its particular contents to customize its view. It also allows you to initialize item elements in ways that can't be done in the declarative form, such as delay-loading images, adding event handlers for specific parts of an item, and optimizing performance.

We'll return to some of these topics later on. For the time being, it's helpful to understand exactly what's going on with declarative templates and how that relates to custom template functions.

Referring to Templates

As I noted before, when you refer to a declarative template in the FlipView or ListView controls, what you're actually referring to is an element, not an element Id. The Id works because the app host creates variables with those names for the elements they identify. However, we don't actually recommend this approach, especially within page controls (which you'll probably use often). The first concern is that only one element can have a particular Id, which means you'll get really strange behavior if you happen to render the page control twice in the same DOM.

The second concern is a timing issue. The element Id variable that the app host provides isn't created until the chunk of HTML containing the element is added to the DOM. With page controls, `WinJS.UI.processAll` is called before this time, which means that element Id variables for templates in that page won't yet be available. As a result, any controls that use an Id for a template will either throw an exception or just show up blank. Both conditions are guaranteed to be terribly, terribly confusing.

To avoid this issue with a declarative template, place the template's name in its `class` attribute:

```
<div data-win-control="WinJS.Binding.Template" class="myItemTemplate" ...></div>
```

Then in your control declaration, use the syntax `select("<selector>")` in the options record, where `<selector>` is anything supported by `element.querySelector`:

```
<div data-win-control="WinJS.UI.ListView"
    data-win-options="{ itemTemplate: select('.myItemTemplate') }"></div>
```

There's more to this, actually, than just a `querySelector` call. The `select` function in the options syntax here searches from the root of its containing page control. If no match is found, it looks for another page control higher in the DOM, then looks in there, continuing the process until a match is found. This lets you safely use two page controls at once that both contain the same class name for different templates, and each page will use the template that's most local.

You can also retrieve the template element using `querySelector` directly in code and assign the result to the `itemTemplate` property. This would typically be done in a page's `ready` function, as demonstrated in the Grid App project, and doing so avoids both concerns identified here because `querySelector` will be scoped to the page contents and will happen after `WinJS.UI.processAll`.

Template Elements and Rendering

The next interesting question about template is this: what, exactly, do we get when instantiating a [WinJS.Binding.Template](#)? This is more or less another WinJS control that turns into an element when you call `WinJS.UI.processAll`. However, it's different in that it removes all its child elements from the DOM, as we know, so it never shows up by itself. It doesn't even have a `winControl` property on its containing element.

What it *does* have, however, is this exceptionally useful function called `render`. Given a data context (an object with properties) and an element, `render` creates a full copy of the template inside the element, resolving any data-binding relationships in the template, both in `data-win-bind` and `data-win-options` attributes, using the data object. In short, think of a declarative template as a set of instructions that the `render` method uses to do all the necessary `createElement` calls along with setting properties and doing data binding.

As shown on the [How to use templates to bind data](#) topic, you can just instantiate and render a template anywhere you want:

```
var templateElement = document.getElementById("templateDiv");
var renderHere = document.getElementById("targetElement");
renderHere.innerHTML = "";

WinJS.UI.process(templateElement).then(function (templateControl) {
    templateControl.render(myDataItem, renderHere);
});
```

It should be wholly obvious that this is exactly what `FlipView` and `ListView` controls do for each item in a given data source. In the case of `FlipView`, it calls its item template's `render` method each time you switch to a different item in the data source. `ListView` iterates over its `itemDataSource` and calls the

item template's `render` for each item, and does something similar for its `groupDataSource` and the `groupHeaderTemplate`.

Template Functions (Part 1): The Basics

Knowing now that a `WinJS.Binding.Template` control is basically just a set of declarative instructions for its `render` method, it should be plainly obvious that you can just create a custom function to do the same job directly. In addition to an element, the `FlipView/ListView` `itemTemplate` properties and the `ListView` `groupHeaderTemplate` property can also accept a renderer function. The controls use `typeof` at run time to determine what you've assigned to these properties, so if you provide a template element, the controls will call its `render` method; if you provide a function, the controls will just call that function for each item that needs to be rendered. This provides a great deal of flexibility to customize the template based on individual item data.

Indeed, a renderer allows you to individually control not only *how* the elements for each item are constructed but also *when*. As such, renderers are the primary means through which you can implement five progressive levels of optimization, especially for `ListView`. Warning! There be promises ahead! Well, we'll save most of that discussion for the end of the chapter, because we need to look at other `ListView` features first. But here let's at least look at the core structure of a renderer that applies to both `FlipView` and `ListView`, which you can see in the [HTML ListView Item Templating](#) and the [HTML ListView Optimizing Performance](#) samples. We'll be drawing code from the latter.

For starters, you can specify a renderer by name in `data-win-options` for both the `FlipView` and `ListView` controls. That function must be marked for processing as discussed in Chapter 4 since it definitely participates in the `WinJS.UI.processAll` operation, so this is a great place to use `WinJS.Utilities.markSupportForProcessing`. Note that if you assign a function to an `itemTemplate` or `groupHeaderTemplate` property in JavaScript, it doesn't need the mark.

In its basic form, a template function receives an item promise as its first argument and returns a promise whose completed handler creates the elements for that item. Huh? Yeah, that confuses me too! So let's look at the basic structure in terms of two functions:

```
function basicRenderer(itemPromise) {
    return itemPromise.then(buildElement);
};

function buildElement (item) {
    var result = document.createElement("div");

    //Build up the item, typically using innerHTML
    return result;
}
```

The renderer is the first function above. It simply says, "When `itemPromise` is fulfilled, meaning the item is available, call the `buildElement` function with that item." By returning the promise from `itemPromise.then`, we allow the collection control that's using this renderer to chain the item promise and the element-building promise together. This is especially helpful when the item data is coming

from a service or other potentially slow feed, and it's very helpful with incremental page loading because it allows the control to cancel the promise chain if the page is scrolled away before those operations complete. In short, it's a good idea!

Just to show it, here's how we'd make a renderer directly usable from markup, as in `data-win-options = "{itemTemplate: Renderers.basic }"`:

```
WinJS.Namespace.define("Renderers", {
    basic: WinJS.Utilities.markSupportedForProcessing(function (itemPromise) {
        return itemPromise.then(buildElement);
    })
});
```

It's also common to just place the contents of a function like `buildElement` directly within the renderer itself, resulting in a more concise expression of the exact same structure:

```
function basicRenderer(itemPromise) {
    return itemPromise.then(function (item) {
        var result = document.createElement("div");

        //Build up the item, typically using innerHTML

        return result;
    });
};
```

What you then do inside the element creation function (whether named or anonymous) defines the item's layout and appearance. Returning to Scenario 8 that we've added to the FlipView sample, we can take this declarative template, where we had to play some tricks to get data binding to work:

```
<div id="pictures_ItemTemplate" data-win-control="WinJS.Binding.Template">
    <div class="overlaidItemTemplate">
        <img class="image" data-win-bind="src: thumbnail InitFunctions.thumbURL;
            alt: name WinJS.Binding.oneTime" />
        <div class="overlay">
            <h2 class="ItemTitle" data-win-bind="innerText: name WinJS.Binding.oneTime"></h2>
        </div>
    </div>
</div>
```

and turn it into the following renderer, keeping the two functions here separate for the sake of clarity:

```
//Earlier: assign the template in code
myFlipView.itemTemplate = thumbFlipRenderer;

//The renderer (see Template Functions (Part 2) later in the chapter for optimizations)
function thumbFlipRenderer(itemPromise) {
    return itemPromise.then(buildElement);
};

//A function that builds the element tree
function buildElement (item) {
```

```

var result = document.createElement("div");
result.className = "overlaidItemTemplate";

var innerHTML = "<img class='thumbImage'>";
var innerHTML += "<div class='overlay'>";
innerHTML += "<h2 class='ItemTitle'>" + item.data.name + "</h2>";
innerHTML += "</div>";

result.innerHTML = innerHTML;

//Set up a listener for thumbnailUpdated that will render to our img element
var img = result.querySelector("img");
WinJS.UI.StorageDataSource.loadThumbnail(item, img).then();

return result;
}

```

Because we have the individual `item` in hand already, we don't need to quibble over the details of declarative data binding and converters: we can just extract the properties we need (from `item.data`) and assign them accordingly. As before, remember that the `thumbnail` property of the `FileInformation` item might not be set yet. This is where we can use the `StorageDataSource.loadThumbnail` method to listen for the `FileInformation.onthumbnailupdated` event. This helper function will render the thumbnail into our `img` element when the thumbnail becomes available (with a little animation to boot!).

You might also notice that I'm building most of the element by using the root `div.innerHTML` property instead of calling `createElement` and `appendChild` and setting individual properties explicitly. Except for very simple structures, setting `innerHTML` on the root element is more efficient because we minimize the number of DOM API calls. This doesn't matter so much for a `FlipView` control whose items are rendered one at a time, but it becomes very important for a `ListView` with potentially thousands of items. Indeed, when we start looking at performance optimizations, we'll also want to render the item in various stages, such as delay-loading images. We'll see all the details in the "Template Functions (Part 2): Promises, Promises!" section at the end of this chapter.

Listview Features and Styling

Having already covered data sources and templates along with a number of `ListView` examples, we can now explore the additional features of the `ListView` control, such as layouts, styling considerations, and cell spanning for multisize items. Optimizing performance then follows in the last section of this chapter. First, however, let me answer a very important question.

When Is ListView the Wrong Choice?

`ListView` is the hands-down richest control in all of Windows. It's very powerful, very flexible, and, as we're already learning, very deep and intricate. But for all that, sometimes it's also just the wrong choice! Depending on the design, it might be easier to just use basic HTML/CSS layout.

Conceptually, a ListView is defined by the relationship between three parts: a data source, templates, and layout. That is, items in a data source, which can be grouped, sorted, and filtered, are rendered using templates and organized with a layout (typically with groups and group headers). In such a definition, the ListView is intended to help visualize a collection of similar and/or related items, where their groupings also have a relationship of some kind.

With this in mind, the following factors strongly suggest that a ListView is a *good* choice to display a particular collection:

- The collection can contain a variable number of items to display, possibly a very large number, showing more when the app runs on a larger display.
- It makes sense to organize and reorganize the items in various groups.
- Group headers help to clarify the common properties of the items in those groups, and they can be used to navigate to a group-specific page.
- It makes sense to sort and/or filter the items according to different criteria.
- Different groupings of items and information about those groups suggest ways in which semantic zoom would be a valuable user experience.
- The groups themselves are all similar in some way, meaning that they each refer to a similar kind of thing. For example, letters, place names, and product categories are similar; a news feed, a list of friends, and a calendar of holidays are not similar.

On the flip side, opposite factors suggest that a ListView is *not* the right choice:

- The collection contains a limited or fixed number of items, or it isn't really a collection of related items at all.
- It doesn't make sense to reorganize the groupings or to filter or sort the items.
- You don't want group headers at all.
- You don't see how semantic zoom would apply.
- The groups are highly dissimilar—that is, it wouldn't make sense for the groups to sit side-by-side if the headers weren't there.

Let me be clear that I'm not talking about *design* choices here—your designers can hand you any sort of layout they want and it's *your* job to implement it! What I'm speaking to is how you choose to try doing that implementation, whether with right controls or just with HTML/CSS layout.

I say this because in working with the developers who created the very first WinRT apps for the Windows Store, we frequently saw them trying to use ListView in situations where it just wasn't needed. An app's huge page, for example, might combine a news feed, a list of friends, and a calendar. An item details page might display a picture, a textual description, and a media gallery. In both cases, the page contains a limited number of sections and the sections contain very different content, which is to say

that there isn't a similarity of items across the groups. Because of this, using a `ListView` is more complicated than just using a single pannable `div` with a CSS grid in which you can lay out whatever sections you need.

Within those sections, of course, you might use `ListView` controls to display an item collection, but for the overall page, a simple `div` is all you need. I've illustrated these choices in Figure 5-7 using an image from the [Navigation design for WinRT apps](#) topic, since you'll probably receive similar images from your designers. Ignoring the navigation arrows, the hub and details pages typically use a `div` at the root, whereas a section page is often a `ListView`. Within the hub and details pages there might be some `ListView` controls, but where there is essentially fixed content (like a single item), the best choice is a `div`.

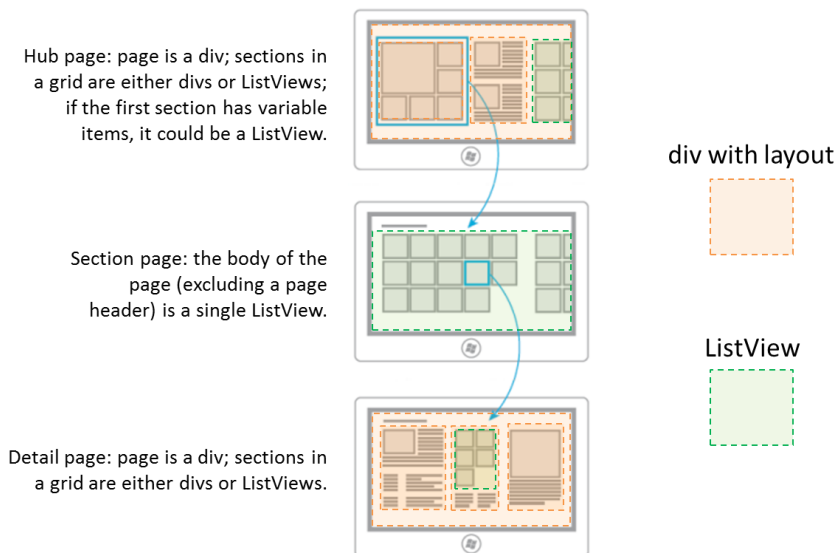


Figure 5-7 Breaking down typical hub-section-detail page designs into `div` elements and `ListView` controls.

A clue that you're going down the wrong path, by the way, is if you find yourself trying to combine multiple collections of unrelated data into a single source, binding that source to a `ListView`, and implementing a renderer to tease all the data apart again so that everything renders properly! All that extra work could be avoided simply by using HTML/CSS layout.

Options, Selections, and Item Methods

In previous sections we've already seen some of the options you can use when creating a `ListView`, options that correspond to the control's properties that are accessible also from JavaScript. Let's look now at the complete set of properties, methods, and events, which I've organized into a few groups—after all, those properties and methods form quite a collection in themselves! Since the details for the individual properties are found on the [WinJS.UI.ListView object](#) reference page, what's most

useful here is to understand how the members of these groups relate:

- **Addressing items** The `currentItem` property gets or sets the item with the focus, and the `elementFromIndex` and `indexOfElement` methods let you cross-reference between an item index and the DOM element for that item. The latter could be useful if you have other controls in your item template and need to determine the surrounding item in an event handler.
- **Item visibility** The `indexOfFirstVisible` and `indexOfLastVisible` properties let you know what indices are visible, or they can be used to scroll the ListView appropriate for a given item. The `ensureVisible` method brings the specified item into view, if it's been loaded. There is also the `scrollPosition` property that contains the distance in pixels between the first item in the list and the current viewable area. Though you can set the scroll position of the ListView with this property, it's reliable only if the control's `loadingState` (see "Loading state" group below) is `ready`, otherwise the ListView may not yet know its actual dimensions. It's thus recommended that you instead use `ensureVisible` or `indexOfFirstVisible` to control scroll position.
- **Item invocation** The `itemInvoked` event, as we've seen, fires when an item is tapped, unless the `tapBehavior` property is not set to `none`, in which case no invocation happens. Other `tapBehavior` values from the [WinJS.UI.TapBehavior enumeration](#) will always fire this event but determine how the item selection is affected by the tap. Do note that you can override the selection behavior on a per-item basis using the `selectionchanging` event and suppress the animation if needed. See the "Tap/Click Behaviors" sidebar after this list.
- **Item selection** The `selectionMode` property contains a value from the [WinJS.UI.-SelectionMode enumeration](#), indicating single-, multi-, or no selection. At all times the `selection` property contains a [ListViewItems object](#) whose methods let you enumerate and manipulate the selected items (such as setting selected items through its `set` method). Changes to the selection fire the `selectionchanging` and `selectionchanged` events; with `selectionchanging`, its `args.detail.newSelection` property contains the newly selected items. For more on this, refer to the [HTML ListView Customizing Interactivity sample](#).
- **Swiping** Related to item selection is the `swipeBehavior` property that contains a value from the [WinJS.UI.SwipeBehavior enumeration](#). "Swiping" is the top-down touch gesture on an item to select it. If this is set to `none`, swiping has no effect on the item and the gesture is bubbled up to the parent elements, allowing a vertically oriented ListView or its surround page to pan. If this is set to `select`, the gesture is processed by the item to select it.
- **Data sources and templates** We've already seen the `groupDataSource`, `groupHeaderTemplate`, `itemDataSource`, and `itemTemplate` properties already. There are two related properties, `resetGroupHeader` and `resetItem`, that contain functions

that the `ListView` will call when recycling elements. This is explained in “Template Functions (Part 2): Promises, Promises!” section.

- **Layout** As we’ve also seen, the `layout` property (an object) describes how items are arranged in the `ListView`, which we’ll talk about more in “Layouts and Cell Spanning” below. We’ve also seen the `forceLayout` function that’s specifically used when a `display: none` style is removed from a `ListView` and it needs to re-render itself.
- **Loading behavior** As explained in the “Optimizing `ListView` Performance” section later on, this group determines how the `ListView` loads pages of items (which is why `ensureVisible` doesn’t always work if a page hasn’t been loaded). When the `loadingBehavior` property is set to `"randomaccess"` (the default), the `ListView`’s scrollbar reflects the total number of items but only five total pages of items (to a maximum of 1000) are kept in memory at any given time as the user pans around. (The five pages are the current page, plus two buffer pages both ahead and behind.) The other value, `"incremental"`, is meant for loading some number of pages initially and then loading additional pages when the user scrolls toward the end of the list (keeping all items in memory thereafter). Incremental loading works with the `automaticallyLoadPages`, `pagesToLoad`, and `pagesToLoadThreshold` properties, along with the `loadMorePages` method, as we’ll see.
- **Loading state** The read-only `loadingState` property contains either `"itemsLoading"` (the list is requesting items and headers from the data source), `"viewportLoaded"` (all items and headers that are visible have been loaded), `"itemsLoaded"` (all remaining nonvisible buffered items have been loaded), or `"complete"` (all items are loaded, content in the templates is rendered, and animations have finished). Whenever this property changes, which is basically whenever the `ListView` needs to update its layout due to panning, the `loadingStateChanged` event fires.
- **Miscellany** The `addEventListener`, `removeEventListener`, and `dispatchEvent` (methods) are the standard DOM methods for handling and raising events. These can be used with any other event that the `ListView` supports, including `contentanimating` that fires when the control is about to run an item entrance or transition animation, allowing you to either prevent or delay those animations. The `zoomableView` property contains the `IZoomableView` implementation as required by semantic zoom (apps will never manipulate this property).

Sidebar: Tap/Click Behavior

When you tap or click an item in a `ListView` with the `tapBehavior` property set to something other than `none`, there’s a little ~97% scaling animation to acknowledge the tap. If you have some items in a list that can’t be invoked (like those in a particular group or ones that you show as disabled because backing data isn’t yet available), they’ll still show the animation because the `tapBehavior` setting applies to the whole control. To remove the animation for any specific item,

you can add the `win-interactive` class to its element within a renderer function, which is a way of saying that the item internally handles tap/click events, even if it does nothing but eat them. If at some later time the item becomes invocable, you can, of course, remove that class.

If you need to suppress selection for an item, add a handler for the ListView's `selection-changing` event and call its `args.detail.preventTapBehavior` method. This works for all selection methods, including swipe, mouse click, and the Enter key.

Styling

Following the precedent of Chapter 4 and the earlier section on ListView, styling is best understood visually as in Figure 5-8, where I've applied some garish CSS to some of the `win-*` styles so that they stand out. I highly recommend you look at the [Styling the ListView and its items](#) topic in the documentation, which details some additional styles that are not shown here.

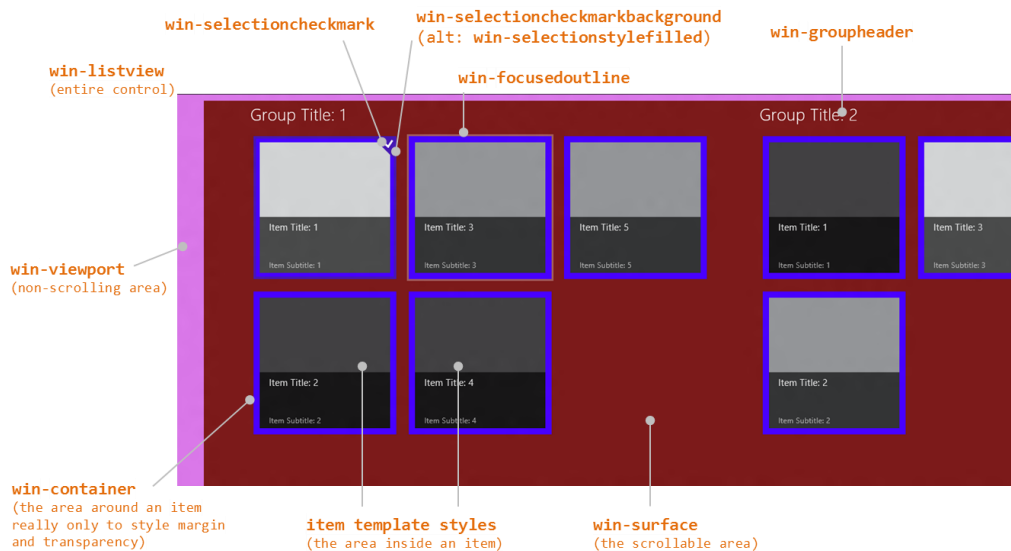


Figure 5-8 Style classes as utilized by the ListView control.

A few notes about styling:

- Remember that Blend is your best friend here!
- As with styling the FlipView, a class like `win-listview` is most useful with styles like margins and padding, since a property like its background color won't actually show through anywhere (unlike `win-viewport` and `win-surface`).
- `win-viewport` styles the nonscrolling background of the ListView and is rarely used, perhaps for a nonscrolling background image. `win-surface` styles the scrolling

background area.

- `win-container` primarily exists for two things. One is to create space between items using `margin` styles, and the other is to override the default background color, often making its background transparent so that the `win-surface` or `win-viewport` background shows through. Note that if you set a *padding* style here instead of `margin`, you'll create areas *around* what the user will perceive as the item that are still invoked as the item. Not good. So always use `margin` to create space between items.
- Though `win-item` is listed as a style, it's deprecated and may be removed in the future: just style the item template directly.
- The documentation points out that styles like `win-container` and `win-surface` are used by multiple WinJS controls. (FlipView uses a few of them.) If you want to override styles for a ListView, be sure to scope your selectors them with other classes like `.win-listview` or a particular control's Id or class.
- The default ListView height is 400px, and the control does *not* automatically adjust itself to its content. You'll almost always want to override that style in CSS or set it from JavaScript when you know the space that the ListView should occupy, as we'll cover in Chapter 6.
- Styles not shown in the figure but described on [Styling the ListView and its items](#) include `win-focusedoutline`, `win-selection`, `win-selected`, `win-selectionborder`, `win-selectionbackground`, and `win-selectionhint`. There is also the `win-selection-stylefilled` class that you add to an item to use a *filled* selection style rather than the default *bordered* style, as shown here:



Backdrops

There is another ListView visual that is a bit like styling but not affected by styling. This is called the backdrop, an effect that's turned on by default when you use the GridLayout. On low-end hardware, especially low-power mobile devices, panning around quickly in a ListView can very easily outpace the control's ability to load and render items. To give the user a visual indication of what they're doing, the GridLayout displays a simple backdrop of item outlines based on the default item size and pans that until such time as real items are rendered. As we'll see in the next section, you can disable this feature with the GridLayout's `disableBackdrop` property and override its default gray color with the `backdropColor` property.

Layouts and Cell Spanning

The ListView's `layout` property, which you can set at any time, contains an object that's used to organize the list's items. WinJS provides two prebuilt layouts: `WinJS.UI.GridLayout` and `WinJS.UI.ListLayout`. The first, already described earlier in this chapter, provides a horizontally panning two-dimensional layout that places items in columns (top to bottom) and then rows (left to right). The second is a one-dimensional top-to-bottom layout, suitable for vertical lists (as in snapped view).

Technically speaking, the `layout` property is an object in itself, containing some number of other properties along with a `type` property. Typically, you see the syntax `layout: { type: <layout> }` in a ListView's `data-win-options` string, where `<layout>` is `WinJS.UI.GridLayout` or `WinJS.UI.ListLayout` (technically, the name of a constructor function). In the declarative usage, layout can also contain options that are specific to the type. For example, the following configures a GridLayout with headers on the left and four rows:

```
layout: { type: WinJS.UI.GridLayout, groupHeaderPosition: 'left', maxRows: 4 }
```

If you create the layout object in JavaScript by using `new` to call the constructor directly (and assigning it to the `layout` property), you can specify additional options with the constructor. This is done in the Grid App project template's `initializeLayout` method in `groupedItems.js`:

```
listView.layout = new ui.GridLayout({ groupHeaderPosition: "top" });
```

You can also set properties on the ListView's `layout` object in JavaScript once it's been created, if you want to take that approach. Changing properties will generally update the layout.

In any case, each layout has its own unique options. For GridLayout, we have these:

- `groupHeaderPosition` controls the placement of headers in relation to their groups; can be set to `"left"` or `"top"`.
- `maxRows` controls the number of items the layout will place vertically before starting another column.
- `backdropColor` provides for customizing the default backdrop color (see "Backdrops" in the previous section), and `disableBackdrops` turns off the effect entirely.
- `groupInfo` identifies a function that returns an object whose properties indicate whether cell spanning should be used and the size of the cell (see below). This is called only once within a layout process.
- `itemInfo` identifies a function for use with cell spanning that returns an object of properties describing the exact size for each item and whether the item should be placed in a new column (see below).

The GridLayout also has a read-only property called `horizontal` that's always `true`. As for the ListLayout, its `horizontal` property is always `false` and has no other configurable properties.

Now, because the `ListView`'s `layout` property is just an object (or the name of a constructor for such an object), can you create a custom layout function of your own? Yes, you can: create a class that provides the same public methods as the built-in layouts, as described by the (currently under-documented) `WinJS.UI.Layout` class. From there the layout object can provide whatever other options (properties and methods) are applicable to it. This topic is somewhat beyond the scope of this chapter, though I hope to cover it in an appendix, blog post, or other such content later on until such time as official documentation exists.

Now before you start thinking that you might need a custom layout, the `GridLayout` provides for something called *cell spanning* that allows you to create variable-sized items (not an option for `ListView`). This is what its `groupInfo` and `itemInfo` properties are for, as demonstrated in Scenarios 4 and 5 of the [HTML ListView Item Templates sample](#) and shown in Figure 5-9.

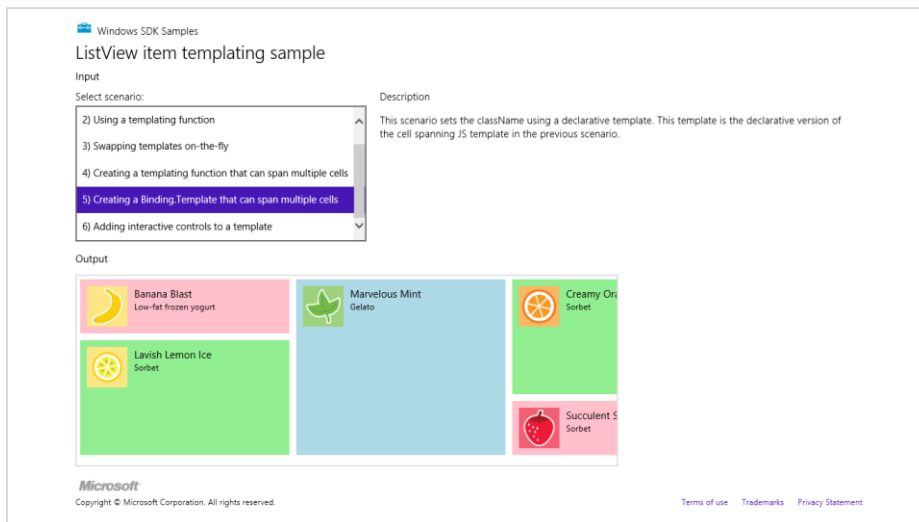


Figure 5-9 The SDK's `ListView` item templates sample showing multisize items through cell spanning.

The basic idea of cell spanning is to define a grid for the `GridLayout` based on the size of the smallest item (including padding and margin styles). For best performance, make the grid as coarse as possible, where every other element in the `ListView` is a multiple of that size.

You turn on cell spanning through the `GridLayout`'s `groupInfo` property. This is a function that returns an object with three properties: `enableCellSpanning`, which should be set to `true`, and `cellWidth` and `cellHeight`, which contain the pixel dimensions of your minimum cell (which, by the way, is what the `GridLayout`'s backdrop feature will use for its effects in this case). In the sample (see `js/data.js`), this function is named `groupInfo` like the layout's property. I've given it a different name here for clarity:

```
function cellSpanningInfo() {
    return {
        enableCellSpanning: true,
```

```

        cellWidth: 310,
        cellHeight: 80
    };
}

```

You then specify this function as part of the `layout` property in `data-win-options`:

```
layout: { type: WinJS.UI.GridLayout, groupInfo: cellSpanningInfo }
```

or you can set `layout.groupInfo` from JavaScript. In any case, once you've announced your use of cell spanning, your item template should set each item's `style.width` and `style.height` properties, plus applicable padding values, to multiples of your `cellWidth` and `cellHeight` according to the following formulae (which are two arrangements of the same formula):

$$\text{templateSize} = ((\text{cellSize} + \text{margin}) \times \text{multiplier}) - \text{margin}$$

$$\text{cellSize} = ((\text{templateSize} + \text{margin}) / \text{multiplier}) - \text{margin}$$

In the sample, these styles are set by assigning each item one of three class names: `smallListItemIconTextItem`, `mediumListItemIconTextItem`, and `largeListItemIconTextItem`, whose relevant CSS is as follows (from `scenario4.css` and `scenario5.css`):

```

.smallListItemIconTextItem {
    width: 300px;
    height: 70px;
    padding: 5px;
}

.mediumListItemIconTextItem {
    width: 300px;
    height: 160px;
    padding: 5px;
}

.largeListItemIconTextItem {
    width: 300px;
    height: 250px;
    padding: 5px;
}

```

Since each of these classes has padding, their actual sizes from CSS are 310x80, 310x170, and 310x260. The margin to apply in the formula comes from the `win-container` style in the WinJS stylesheet, which has a margin of 5px. Thus:

$$((80 + 10) * 1) - 10 = 80; \text{ minus 5px padding top and bottom} = \text{a height of 70px in CSS}$$

$$((80 + 10) * 2) - 10 = 170; \text{ minus 5px padding} = \text{height of 160px}$$

$$((80 + 10) * 3) - 10 = 260; \text{ minus 5px padding} = \text{height of 250px}$$

The only difference between Scenario 4 and Scenario 5 is that the former assigns class names to the items through a template function. The latter does it through a declarative template and data-binds the class name to an item data field containing those values.

As for the `itemInfo` function, this is a way to optimize the performance of a ListView when using cell spanning. Without assigning a function to this property, the GridLayout has to manually determine the width and height of each item as it's rendered, and this can get slow if you pan around quickly with a large number of items. Since you probably already know item sizes yourself, you can return that information through the `itemInfo` function. This function receives an item index and returns an object with the item's `width` and `height` properties. (We'll see a working example in a bit.)

```
function itemInfo(itemIndex) {
    //determine values for itemWidth and itemHeight given itemIndex
    return {
        newColumn: false,
        itemWidth: itemWidth,
        itemHeight: itemHeight
    };
}
```

Clearly, this function will be called for every item in the list but only if cell spanning has been turned on through the `groupInfo` function. Again, unless your list is relatively small, you'll very much improve performance by providing item dimensions through this function.

You probably also noticed that `newColumn` property in the return value. As you might guess, this instructs the GridLayout to start a new column with this item, allowing you to control that particular behavior. You can even use `newColumn` by itself, if you like, with a smallish list.

Now you might be asking: what happens if I set different sizes in my item template but don't actually announce cell spanning? Well, you'll end up with overlapping (and rather confusing) items. This is because *the GridLayout takes the first item in a group as the basic measure for the rest of the items* (and the backdrop grid as well; it does *not* attempt to automatically size each item according to content). Try this with Scenarios 4 or 5: remove the `layout.groupInfo` property from the ListView's `data-win-options` in `scenario4.html` or `scenario5.html` and restart the app, and you'll see the medium and large items bleeding into those that follow:



Then go into `data.js`, set the first item's style in the `myCellSpanningData` array to be `largeListIconTextItem`, and restart; the ListView then does layout with that as the basic item size:



Using the first item's dimension like this underscores the fact that a ListView with cell spanning will take more time to render because it must measure each item as it gets laid out, with or through the `itemInfo` function. For this reason, cell spanning is probably best avoided for large lists.

Where all this gets a little more interesting, which the sample doesn't show, is how the GridLayout deals with items that vary in width. Its basic algorithm is still to lay out columns from top to bottom and then left to right, but it will now infill empty spaces next to smaller items when larger ones create those gaps. To demonstrate this, let's modify the sample so that the smallest item is 155x80 (half the original size), the medium item is 310x80, and the large item is 310x160. Here are the changes to make that happen:

1. Undo any changes from the previous tests: in `scenario4.html`, add `groupInfo` back to `data-win-options`, and in `data.js`, change the class in the first item of `myCellSpanningData` back to `smallListItemIconItem`.
2. In `data.js`, change the `cellWidth` in `groupInfo` to 155 (half of 310) and leave `cellHeight` at 80. For clarity, also insert an incrementing number at the start of each item's text in `myCellSpanningData` array.
3. In `scenario4.css`:
 - a. Change the width of `smallListItemIconItem` to 145px. Applying the formula, $((145+10) * 1) - 10 = 145$. Height is 70px.
 - b. Change the width of `mediumListItemIconItem` to 310px and the height to 70px.
 - c. Change the width of `largeListItemIconItem` to 310px and the height to 160px. Applying the formula to the height, $((80+10) * 2) - 10 = 170$ px.
 - d. Set the `width` style in the `#listview` rule to 800px and the `height` to 600px (to make more space in which to see the layout).

I recommend making these changes in Blend where your edits are reflected more immediately than running the app from Visual Studio. In any case, the results are shown in Figure 5-10 where the numbers show us the order in which the items are laid out (and apologies for clipping the text...experiments must make sacrifices at times!). A copy of the SDK sample with these modifications is also given in the companion content for this chapter.



Figure 5-10 Modifying the SDK's ListView item templates sample to show cell spanning more completely.

In the modified sample I've also included an `itemInfo` function in `data.js`, as you may have already noticed. It returns the item dimensions according to the type specified for the item:

```
function itemInfo(index) {
    //getItem(index).data retrieves the array item from a WinJS.Binding.List
    var item = myCellSpanningData.getItem(index).data;
    var width, height;

    switch (item.type) {
        case "smallListIconTextItem":
            width = 145;
            height = 70;
            break;

        case "mediumListIconTextItem":
            width = 310;
            height = 70;
            break;

        case "largeListIconTextItem":
            width = 310;
            height = 160;
            break;
    }

    return {
        newColumn: false,
        itemWidth: width,
        itemHeight: height
    };
}
```

You can set a breakpoint in this function and verify that it's being called for every item; you can also

see that it produces the same results. Now change the return value of `newColumn` as follows, to force a column break before item #7 and #15 in Figure 5-10, because they oddly span columns:

```
newColumn: (index == 6 || index == 14), //Break on items 7 and 15 (index is 6 and 14)
```

The result of this change is shown in Figure 5-11.



Figure 5-11 Using new columns in cell spanning on items 7 and 15.

One last thing I noticed while playing with this sample is that if the item size in a style rule like `smallListIconTextItem` ends up being smaller than the size of a child element, such as `.regularListIconTextItem` (which includes margin and padding), the larger size wins in the layout. As you experiment, you might also want to remove the default 5px margin that's set for `win-container`. This is what creates the space between the items in Figure 5-10 but has to be added into the equations. The following rule will set that margin to 0px:

```
#listView > .win-horizontal .win-container {
  margin: 0px;
}
```

Optimizing ListView Performance

I've often told people that there's so much you can do and learn about ListView that it could be a book in itself! Indeed, it would have been easy for Microsoft to have just created a basic control that let you create templated items and have left it at that. However, knowing that the ListView would be utterly central to a large number of apps (perhaps the majority outside the gaming category), and expecting that the ListView would be called upon to host thousands or even tens of thousands of items, a highly skilled and passionate group of engineers has gone to great extremes to provide many levels of sophistication that will help your apps perform their best.

One optimization is the ability to demand-load pages of items as determined by the `ListView's loadingBehavior` property, as described in the next two sections. The other optimization is to use template functions to delay-load different parts of each item, such as images, as well as to defer actions like animations until an item actually becomes visible, which is covered in the third section below. In all cases, the whole point of these optimizations is to help the `ListView` display the most important items or parts of items as quickly as possible, deferring the loading and rendering of other items or less important item elements until they're really needed.

I did want to point out that the [Using ListView](#) topic in the documentation contains even more suggestions than I'm able to include here. (I do have other chapters to write!) I encourage you to study that topic as well, and who knows—maybe you'll be the one to write the complete `ListView` book! Furthermore, additional guidance on appwide performance can be found on [Performance best practices for WinRT apps using JavaScript](#), which contains the `Using ListView` topic.

Random Access

If you're like myself and others in my family, you probably have an ever-increasing stockpile of digital photographs that make you glad that 1TB+ hard drives keep dropping in price. In other words, it's not uncommon for many consumers to have ready access to collections of tens of thousands of items that they will at some point want to pan through in a `ListView`. But just imagine the overhead of trying to load every thumbnails for every one of those items into memory to display in a list. On low-level hardware, you'd probably be causing every suspended app to be quickly terminated, and the result will probably be anything but "fast and fluid"! The user might end up waiting a *really* long time for the control to become interactive and will certainly get tired of watching a progress ring!

With this in mind, the default `loadingBehavior` property for a `ListView` is set to `"randomaccess"`. In this mode, the `ListView's` scrollbar will reflect the total extent of the list so that the user has some idea of its size, but the `ListView` keeps a total of only five pages or screenfuls of items in memory at any given time (with an overall limit of 1000 items). For most pages, this means the visible page (in the viewport) plus two buffer pages ahead and behind. (If you're viewing the first page, the buffer extends four pages ahead; if you're on the last page, the buffer extends four pages behind—you get the idea.)

Whenever the user pans to a location in the list, any pages that fall out of the viewport or buffer zone are discarded (almost—we'll come back to this in a moment), and loading of the new viewport page and its buffer pages begins. Thus the `ListView's loadingState` property will start again at `itemsLoading`, then to `viewportLoaded` when the visible items are rendered, then `itemsLoaded` when the buffered pages are loaded, and then `complete` when everything is done. Again, at any given time, only five pages of items are loaded into memory.

Now when I said that previously loaded items get discarded when they move out of the viewport/buffer range, what actually happens is that the items get *recycled*. One of the most expensive parts of rendering an item is creating its DOM elements, so the `ListView` actually takes those elements, moves them to a new place in the list, and fills them in with new content. This will become important when we look at optimization in template functions shortly.

Incremental Loading

Apart from potentially very large but known collections, other collections are, for all intents and purposes, essentially unbounded, like a news feed that might have millions of items stretching back to the Cenozoic Era (at least by Internet reckoning!). With such collections, you probably won't know just how many items there are at all; the best you can really do is just load another chunk when the user wants them.

This is what the `loadingBehavior` of "incremental" is for. In this mode, the `ListView`'s scrollbar will reflect only what's loaded in the list, but if the user passes a particular threshold—for instance, they pan to the end of the list—the `ListView` will ask the data source for more pages of items and add them to the list, updating the scrollbar, of course. In this case, all of the loaded items remain loaded, providing for very quick panning within the loaded list but with potentially more memory consumption than random access.

The incremental loading behavior is demonstrated in Scenarios 2 and 3 of the [ListView loading behaviors sample](#). (Scenario 1 covers random access, but it's nothing different than we've already seen.) Incremental loading activates the following characteristics:

- The `ListView`'s `pagesToLoad` property indicates how many pages or screenfuls of items get loaded at a time. The default value is 5.
- The `automaticallyLoadPages` property indicates whether the `ListView` should load new pages automatically as you pan through the list. If `true` (the default), as demonstrated in Scenario 2, as you pan toward the end of the list you'll see the scrollbar change as new pages are loaded. If `false`, as demonstrated in Scenario 3, new pages are not loaded until you call the `loadMorePages` method directly.
- When `automaticallyLoadPages` is `true`, the `pagesToLoadThreshold` property indicates how close the user can get to the current end of the list before new page loads are triggered. The default value is 2.
- When new pages start to load (either automatically or in response to `loadMorePages`), the `ListView` will start updating the `loadingState` property firing `loadingstatechanged` events as described already.

Template Functions (Part 2): Promises, Promises!

What we just discussed with the `ListView`'s loading behavior options pertained to the incremental loading of pages. It's helpful now to combine this with incremental loading of *items*. For that, we need to look at what's sometimes referred to as the *rendering pipeline* as implemented in template functions.

When we first looked at template functions earlier (see "How Templates Really Work"), I noted that they give us control over both how and when items are constructed and that such functions—again, called renderers—are *the* means through which you can implement five progressive levels of optimization for `ListView` (and `FlipView`, though this is less common). Just using a renderer, as we

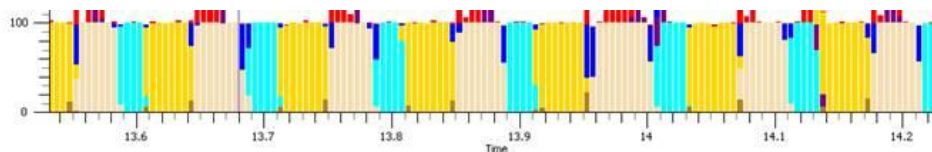
already saw, is level 1; now we're ready to see the other four levels. This is a fascinating subject, because it shows the kind of sophistication that the ListView has implemented for us!

Our context for this discussion is the [HTML ListView optimizing performance](#) sample that demonstrates all these levels and allows you to see their respective effects. Here's an overview:

- A *simple* or basic renderer allows control over the rendering on a per-item basis.
- A *placeholder* renderer separates creation of the item element into two stages. The first stage returns only those elements that define the shape of the item. This allows the ListView to quickly do its overall layout before all the details are filled in, especially when the data is coming from a potentially slow source. When item data is available, the second stage is then invoked to copy that data into the item elements and even creating additional elements therein.
- A *recycling placeholder* renderer adds the ability to reuse an existing chunk of DOM for the item, which is much faster than having to create one from scratch. For this purpose, the ListView, knowing that it will be frequently paged around, holds onto some number of item elements when they go offscreen. In your renderer, then, you add a code path to clean up a recycled element if one is given to you, and return that as your placeholder.
- A *multistage* renderer extends the recycling renderer both to delay-load images and other media until the item element is fully built up in the ListView and also to delay any visibility-related actions, such as animations, until the item is actually on-screen.
- Finally, a multistage *batching* renderer adds the ability to add images and other media as a batch, thereby rendering and possibly animating their entrance into the ListView as a group such that the system's GPU can be employed more efficiently.

With all of these renderers, you should strive to make them execute as fast as possible. Especially minimize the use of DOM API calls, which includes setting individual properties. Use an [innerHTML](#) string where you can to create elements rather than discrete calls, and minimize your use of [getElementById](#), [querySelector](#), and other DOM-traversal calls by caching the elements you refer to most often. This will make a big difference.

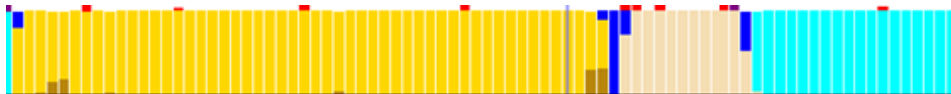
To visualize the effect of these improvements, the following graphic shows an example of how typical ListView rendering typically happens:



The yellow bars indicate execution of the app's JavaScript—that is, time spent inside the renderer. The manila bars indicate the time spent in DOM layout, and aqua bars indicate actual rendering to the

screen. As you can see, when elements are added one by one, there's quite a bit of breakup in what code is executing when, and the kicker here is that most display hardware refreshes only every 16 milliseconds. As a result, there's lots of choppiness in the visual rendering.

After making improvements, the chart can look like the one below, where the app's work is combined in one block, thereby significantly reducing the DOM layout process (the manila):



As for all the other little bits in these graphics, they come from the performance tool called XPerf that's part of the Windows SDK (see sidebar), and I haven't studied the details yet. What ultimately matters is that we understand the steps you can take to achieve these ends—namely, the different forms of renderers that you can employ as demonstrated in the sample.

Sidebar: XPerf and msWriteProfilerMark

The XPerf tool in the Windows SDK, which is documented on [Windows Performance Analysis Tools](#), can very much help you understand how your app really behaves on a particular system. Among other things, it logs calls you make to [msWriteProfilerMark](#), as you'll find sprinkled throughout the WinJS source code itself. For these to show up in xperf, however, you need to start logging with this command:

```
xperf -start user -on PerfTrack+Microsoft-IE:0x1300
```

and end logging with this one:

```
xperf -stop user -d <trace_filename>.etl
```

Launching the .etl file you save will run the Windows Performance Analyzer and show a graph of events. Right-click the graph, and then click "Summary Table". In that table, expand Microsoft-IE and then look for and expand the Mshtml_DOM_CustomSiteEvent node. The Field3 column should have the text you passed to [msWriteProfilerMark](#), and the Time(s) column will help you determine how long actions took.

As our baseline, here is a *simple renderer*:

```
function simpleRenderer(itemPromise) {
    return itemPromise.then(function (item) {
        var element = document.createElement("div");
        element.className = "itemTempl";
        element.innerHTML = "<img src='" + item.data.thumbnail +
            "' alt='Databound image' /><div class='content'>" + item.data.title + "</div>";
        return element;
    });
}
```


Again, this structure waits for the item data to become available, and it returns a promise for the element that will be fulfilled at that time.

A *placeholder renderer* separates building the element into two stages. The return value is an object that contains a minimal placeholder in the `element` property and a `renderComplete` promise that does the rest of the work when necessary:

```
function placeholderRenderer(itemPromise) {
    // create a basic template for the item which doesn't depend on the data
    var element = document.createElement("div");
    element.className = "itemTempl";
    element.innerHTML = "<div class='content'>...</div>";

    // return the element as the placeholder, and a callback to update it when data is available
    return {
        element: element,

        // specifies a promise that will be completed when rendering is complete
        // itemPromise will complete when the data is available
        renderComplete: itemPromise.then(function (item) {
            // mutate the element to include the data
            element.querySelector(".content").innerText = item.data.title;
            element.insertAdjacentHTML("afterBegin", "<img src='" +
                item.data.thumbnail + "' alt='Databound image' />");
        })
    };
}
```

The `element` property, in short, defines the item's shape and is returned immediately from the renderer. This lets the `ListView` do its layout, after which it will fulfill the `renderComplete` promise. You can see that `renderComplete` essentially contains the same sort of thing that a simple renderer returns, minus the already created placeholder elements. (For another example, the added Scenario 8 of the `FlipView` example in this chapter's companion content has commented code that implemented this approach.)

A *recycling placeholder* renderer now adds awareness of a second parameter called `recycled` that the `ListView` (but not the `FlipView`) can provide to the function when its `loadingBehavior` is set to `"randomaccess"`. If `recycled` is given, you can just clean out the element, return it as the placeholder, and then fill in the data values within the `renderComplete` promise as before. If it's not provided (as when the `ListView` is first created or when `loadingBehavior` is `"incremental"`), you'll create the element anew:

```
function recyclingPlaceholderRenderer(itemPromise, recycled) {
    var element, img, label;
    if (!recycled) {
        // create a basic template for the item which doesn't depend on the data
        element = document.createElement("div");
        element.className = "itemTempl";
        element.innerHTML = "<img alt='Databound image' style='visibility:hidden;' /><div
class='content'>...</div>";
    }
}
```

```

else {
  // clean up the recycled element so that we can re-use it
  element = recycled;
  label = element.querySelector(".content");
  label.innerHTML = "...";
  img = element.querySelector("img");
  img.style.visibility = "hidden";
}
return {
  element: element,
  renderComplete: itemPromise.then(function (item) {
    // mutate the element to include the data
    if (!label) {
      label = element.querySelector(".content");
      img = element.querySelector("img");
    }
    label.innerText = item.data.title;
    img.src = item.data.thumbnail;
    img.style.visibility = "visible";
  })
};
}

```

In `renderComplete`, be sure to check for the existence of elements that you don't create for a new placeholder, such as `label`, and create them here if needed.

If you'd like to clean out recycled items, you can also provide a function to the `ListView`'s `resetItem` property that would contain the same code as shown above for that case. The same is true for the `resetGroupHeader` property, because you can use template functions for group headers as well as items. We haven't spoken of these as much because group headers are far fewer and don't typically have the same performance implications. Nevertheless, the capability is there.

Next we have the *multistage renderer*, which extends the recycling placeholder renderer to delay-load images and other media until the rest of the item is wholly present in the DOM, and to further delay effects like animations until the item is truly on-screen.

The hooks for this are three methods called `ready`, `loadImage`, and `isOnScreen` that are attached to the item provided by the `itemPromise`. The following code shows how these are used:

```

renderComplete: itemPromise.then(function (item) {
  // mutate the element to update only the title
  if (!label) { label = element.querySelector(".content"); }
  label.innerText = item.data.title;

  // use the item.ready promise to delay the more expensive work
  return item.ready;
  // use the ability to chain promises, to enable work to be cancelled
}).then(function (item) {
  //use the image loader to queue the loading of the image
  if (!img) { img = element.querySelector("img"); }
  return item.loadImage(item.data.thumbnail, img).then(function () {
    //once loaded check if the item is visible

```

```

        return item.isOnScreen();
    });
}).then(function (onscreen) {
    if (!onscreen) {
        //if the item is not visible, then don't animate its opacity
        img.style.opacity = 1;
    } else {
        //if the item is visible then animate the opacity of the image
        WinJS.UI.Animation.fadeIn(img);
    }
})
})

```

I warned you that there would be promises aplenty in these performance optimizations! But all we have here is the basic structure of chained promises. The first async operation in the renderer updates simple parts of the item element, such as text. It then returns the promise in `item.ready`. When that promise is fulfilled—or, more accurately, *if* that promise is fulfilled—you can use the item's async `loadImage` method to kick off an image download, returning the `item.isOnScreen` promise from that completed handler. When and if that `isOnScreen` promise is fulfilled, you can perform those operations that are relevant only to a visible item.

I emphasize the *if* part of all this because it's very likely that the user will be panning around within the ListView while all this is happening. Having all these promises chained together makes it possible for the ListView to cancel the async operations any time these items are scrolled of view and off any buffered pages. Suffice it to say that the ListView control has gone through a *lot* of performance testing!

Which brings us to the final multistage *batching* renderer, which combines the insertion of images in the DOM to minimize layout and repaint work. In the sample, this uses a function called `createBatch` that utilizes `WinJS.Promise.timeout` method with a 64-millisecond period to combine the image-loading promises of the multistage renderer. Honestly, you'll have to trust me on this one, because you really have to be an expert in promises to understand how it works!

```

//During initialization (outside the renderer)
thumbnailBatch = createBatch();

//Within the renderComplete chain

//...
}).then(function () {
    return item.loadImage(item.data.thumbnail);
}).then(thumbnailBatch())
).then(function (newimg) {
    img = newimg;
    element.insertBefore(img, element.firstChild);
    return item.isOnScreen();
}).then(function (onscreen) {
//...

```

Did I warn you about there being promises in your future? Well, fortunately, we've now exhausted

the subject of template functions, but it's time well spent because optimizing ListView performance, as I said earlier, will greatly improve consumer perception of apps that use this control.

What We've Just Learned

- In-memory collection data is managed through `WinJS.Binding.List`, which integrates nicely with collection controls like FlipView and ListView. In-memory collections can come from sources like `WinJS.xhr` and data loaded from files.
- The `WinJS.UI.FlipView` control displays one item at a time; `WinJS.UI.ListView` displays multiple items according to a specific layout.
- Central to both controls is the idea that there is a data source and an item template used to render each item in that source. Templates can be either declarative or procedural.
- ListView works with the added notion of layout. WinJS provides two built-in layouts. GridLayout is a two-dimensional, horizontally panning list; ListLayout is for a one-dimensional vertically panning list. It is possible to implement custom layouts, but that is beyond the scope of this book at present.
- ListView also provides the capability to display items in groups; `WinJS.BindingList` provides methods to create grouped, sorted, and filtered projections of items from a data source.
- The Semantic Zoom control (`WinJS.UI.SemanticZoom`) provides an interface through which you can switch between two different views of a data source, a zoomed-in (details) view and a zoomed-out (summary) view. The two views can be very different in presentation but should display related data. The `IZoomableView` interface is required on each of the views so that the Semantic Zoom control can switch between them and scroll to the correct item.
- WinJS also provides a `StorageDataSource` to create a collection over items from the file system.
- It is also possible to implement custom data sources, as shown by samples in the Windows SDK.
- Procedural templates are implemented as template function, or renderers. These functions can implement progressive levels of optimization for delay-loading images and adding items to the DOM in batches.
- Both FlipView and ListView controls provide a number of options and styling capabilities. ListView also provides for item selection and different selection behaviors.

- The ListView control provides built-in support for optimizing random access of large data sources, as well as incremental access of effectively unbounded data sources.
- The ListView control supports the notion of cell spanning in its GridLayout to support items of variable size, which should all be multiples of a basic cell size.
- Template functions used for item rendering in a ListView provides extensive opportunities for loading and rendering optimization, thereby helping the ListView to perform its best.

Chapter 6

Layout

Compared to other members of my family, I seem to need the least amount of sleep and am often up late at night or up before dawn. To avoid waking the others, I generally avoid turning on lights and just move about in the darkness (and here in the rural Sierra Nevada foothills, it can get *really* dark!). Yet because I know the layout of the house and the furniture, I don't need to see much. I only need a few reference points like a door frame, a corner on the walls, or the edge of the bed to know exactly where I am. What's more, my body has developed a muscle memory for where doorknobs are located, how many stairs there are, how many steps it takes to get around the bed, and so on. It's really helped me understand how visually impaired people "see" their own world.

If you observe your own movements in your home and your workplace—probably when the spaces are lit!—you'll probably find that you move in fairly regular patterns. This is actually one of the most important considerations in home design: a skilled architect looks carefully at how people in the home might move between primary spaces like the kitchen, dining room, and living room, and even within a single workspace like the kitchen. Then they design the home's layout so that the most common patterns of movement are easy and free from obstructions. If you've ever lived in a home where it *wasn't* designed this way, you can very much appreciate what I'm talking about!

The two key points here are these: first, good layout makes a huge difference in the usability of any space, and second, human beings quickly form habits around how they move about within a space, habits that hopefully make their movement more efficient and productive.

Good app design clearly follows the same principles, which is exactly why Microsoft recommends following consistent patterns with your apps, as described on [Designing UX for apps](#) and [Design guidance for WinRT apps](#). Those recommendations are not in any way whimsical or haphazard: they are the result of many years of research and investigation into what would really work best for apps and for Windows 8 as a whole. The placement of the charms, for instance, as well as commands on an app bar (as we'll see in Chapter 7, Commanding UI"), arise from the reality of human anatomy, namely how far we can move our thumbs around the edges of the screen when holding a tablet device.

With page layout, in particular, the recommendations on [Understanding the Windows 8 silhouette](#)—about where headers and body content are ideally placed, the spacing between items, and so forth—can seem rather limiting, if not draconian. The silhouette, however, is meant to be a good starting point but not a hard-and-fast rule. What's most important is that the shape of an app's layout helps users develop a visual muscle memory that can be applied across many apps. Research showed that users will actually develop such habits very quickly, even within a matter of minutes, but of course those habits are not exact to specific pixels! In other words, the silhouette represents a general shape that helps users immediately understand how an app works and where to go for what functions, just like you can easily recognize the letter "S" in many different fonts. This is very efficient and productive.

On the other hand, when presented with an app that used a completely different layout (or worse, a layout that was similar to the silhouette but behaved differently), users must expend much more energy just figuring out where to look and where to tap, just as I would have to be much more careful late at night if you moved all my furniture around!

The bottom line is that there are very good reasons behind *all* the WinRT app design recommendations, layout included. As I've said before, if you're fulfilling the designer role for your app, study the guidelines referred to above. If someone else is fulfilling that role, make sure *they* study the guidelines! Either way, we'll be reviewing the key principles in the first section of this chapter.

After that, our focus will be on how we implement layout designs, not creating the designs themselves. (While I apparently got the mix of my parent's genes that bestowed an aptitude for technical communication, my brother got the most of the genes for artistry!) For example, how does an app respond to view state changes to show the correct page design (for full-screen landscape, filled, snapped, and portrait)? How does the app handle varying display sizes and varying pixel densities?

We'll also spend a little time with the CSS grid and a few other CSS layout features like flexbox and multicolumn text. Generally speaking, these are all CSS standards, so I expect that you already know quite a bit about them or can find full documentation elsewhere.³³ We'll thus cover the basics only briefly, spending more time understanding how these features are best applied within an app and those aspects that are unique to the Windows 8 environment (such as what are called *snap points* on a pannable/scrollable `div`).

I'll remind you again that there still other UI elements like the app bar and flyouts that don't participate in layout at all; I'll cover these in other chapters. There are also auxiliary app pages that service contracts (such as Search and Settings) and exist outside your main navigation flow. These will employ the same layout principles covered in this chapter, but how and when they appear will also be covered later.

Principles of WinRT app Layout

Layout is truly one of the most important considerations in WinRT app design. The principle of "content before chrome" means that most of what you display on any given app page is content, with little in the way of commanding surfaces, persistent navigation tabs, and passive graphical elements like separators, blurs, and gradients that don't in themselves mean anything. Another way of putting this is that content itself should be interactive rather than a passive element that is acted upon when the user invokes some other command. Semantic zoom is a good example of such interactive content—instead of needing buttons or menus elsewhere in the app to switch between views, the

³³ The specifications can be found on <http://www.w3c.org>; specifically start with <http://www.w3.org/standards/webdesign/htmlcss> for both. I also highly recommend the well-designed and curated resources from [Smashing Magazine](http://smashingmagazine.com) for learning the nuances of CSS, which I must admit still seems like voodoo to me at times.

capability is inherent in the control itself, with the small zoom button appearing only when needed and only for mouse users. Other app commands, for the most part, are similarly placed on UI surfaces that appear when needed through app bars and other flyouts, as we'll see in Chapter 7.

In short, "content before chrome" means helping the user be immersed in the experience of the content rather than distracted by nonessentials. In WinRT app design, then, emphasis is given to the space around and between content, which serves to organize and group content without the need for lines and boxes. These essentially transparent "space frames" help consumer's eyes focus on the content that really matters. WinRT app design also uses typography (font size, weight, color, etc.) to convey a sense of structure, hierarchy, and relative importance of different content. That is, since the text on a page is already content, why not use its characteristics—the typography—to communicate what used to be done with extraneous chrome? (As with the layout silhouette, the general use of the Segoe UI font within WinRT app design is not a hard-and-fast requirement, but a starting point. Having a consistent *type ramp* for different headings is more important than the font.)

As an example, Figure 6-1 shows a typical desktop or web application design for an RSS reader. Notice the persistent chrome along the top and bottom: search commands, navigation tabs, navigation controls, and so forth. This takes up perhaps 20% of the screen space. In what remains, nearly two-thirds is taken up by organizational elements, leaving 20-25% of the screen space for the content we actually care about, which is the article.

Figure 6-2 shows a WinRT app design for the same content. Notice how all the ancillary commands have been moved offscreen. Search would be accomplished through the Search charm; Settings through the Settings charm; adding feeds, refresh, and navigation through commands on to the app bar; and switching views through semantic zoom. Typography is used to convey the hierarchy instead of a folder control, which then leaves the bulk of the display—nearly 75%—for the content. As a result, we can see much more of that content than before, which creates a much more immersive and engaging experience, don't you think?

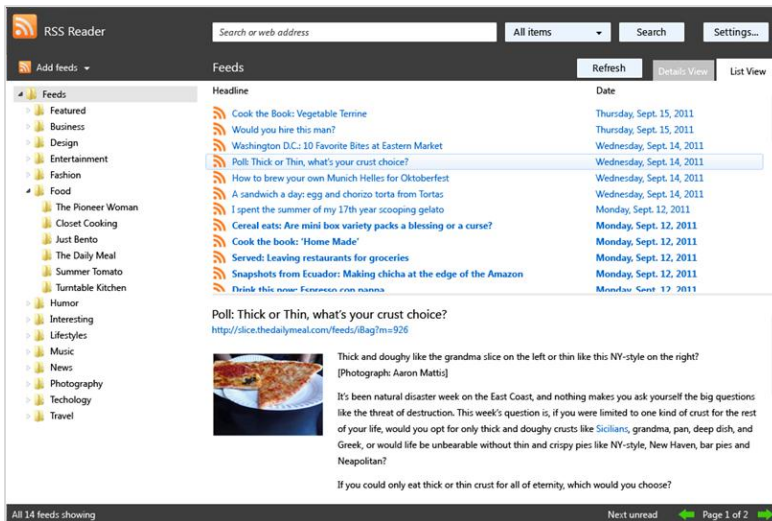


Figure 6-1 A typical desktop or web application design that emphasizes chrome at the expense of content.

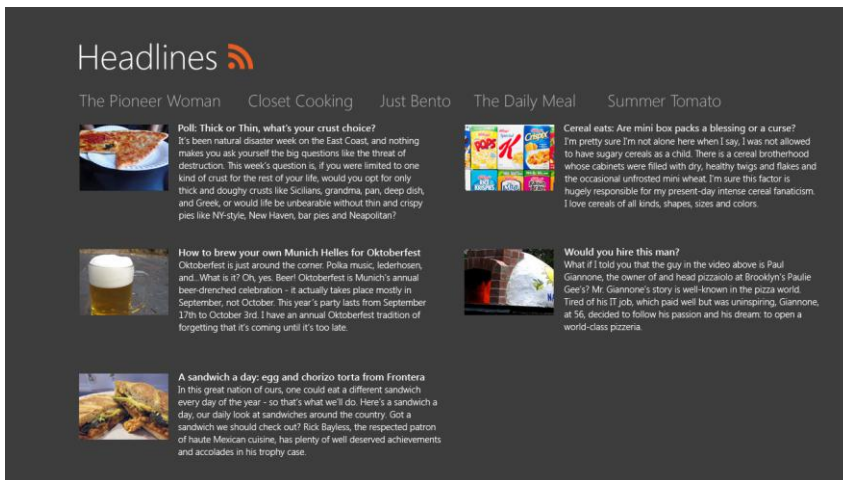


Figure 6-2 The same app as Figure 6-1 reimagined with WinRT app design, where most of the chrome has disappeared, leaving much more space for content.

Even where typography is concerned, WinRT app design encourages the use of distinct font sizes, called the typographic ramp, to establish a sense of hierarchy. The default WinJS stylesheets—`ui-light.css` and `ui-dark.css`—provide four fixed sizes where each level is proportionally larger than the previous (42pt = 80px, 20pt = 40px, etc.), as shown in Figure 6-3. These proportions allow users to easily establish an understanding of content structure with just a glance. Again, it's a matter of encouraging habit and muscle memory, and Microsoft's research has shown that beyond this size granularity, users are generally unable to differentiate where a piece of content fits in a hierarchy.

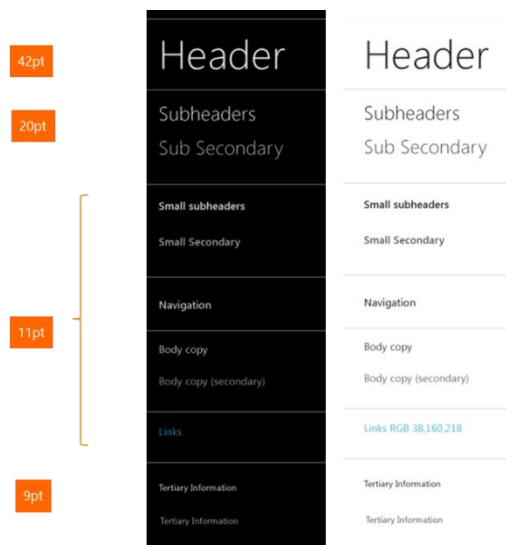


Figure 6-3 The typographic ramp of WinRT app design, shown in both the `ui-dark.css` (left) and `ui-light.css` (right) stylesheets.

Within the body of content, then, WinRT app design encourages these layout principles:

- Let content flow from edge to edge.
- Keep ergonomics in mind: pan along the long edge of the view (primarily horizontal in landscape views, vertical in snapped view and possibly portrait).
- Pan on a single axis only to create a sense of stability and to support swiping to select (as with the `ListView` controls).
- Create visual alignment, structure, and clarity with the WinRT app silhouette, aligning elements on a grid for consistency. Refer again to [Understanding the Windows 8 silhouette](#). This shape is what allows a consumer's eyes to recognize something as a WinRT app without having to think about it, which provides a feeling of familiarity and confidence.

As I've mentioned before, the project templates in Visual Studio and Blend have these principles baked right in and thus provide a convenient starting point for apps. Even if you start with the Blank App template, the others like the Grid App will serve as a reference point. This is exactly what we did with the Here My Am! app in Chapter 2, "Quickstart."

The other important guiding principle that's relevant to layout is "snap and scale beautifully." This means making sure you design every page in your app to handle all four view states and to be appropriately adaptive across different display resolutions and pixel densities. We'll look at this subject in the "View States and the Many Faces of Your Display" section below. First, however, let's look at a little piece of core layout code.

Quickstart: Pannable Sections and Snap Points

In Chapter 5, “Collections and Collection Controls,” we spent a little time looking at when a ListView control was the right choice and when it wasn’t. One of the primary cases where developers have inappropriately used a ListView is to implement a home or hub page that contains a variety of distinct content groups arranged in columns, as shown in Figure 6-4 and explained on [Navigation design for WinRT apps](#). At first glance this might look like a ListView, but because the data it’s representing really isn’t a collection and is just a layout of fixed content, it makes sense to use tried-and-true HTML and CSS for the job!



Figure 6-4 The layout of a typical home or hub page of a WinRT app with a fixed header (1), a horizontally pannable section (2), and content sections or categories (3).

I point this out because with all the great controls that WinJS provides, it’s easy to forget that everything you know about HTML and CSS still applies in WinRT apps. After all, those controls are in themselves just blocks of HTML and CSS with some additional methods, properties, and events.

Laying Out the Hub

Let’s see how we’d use plain HTML and CSS to implement the pannable section of the hub page in Figure 6-4. Referring first to [Understanding the Windows 8 silhouette](#), we know that the padding between groups should be four units of 20px each, or 80px. Most of the groups themselves should be square, except for the second one which is only half the width. On a baseline 1366x768 display, the height of each section would be 768px minus 128px (for the header) minus the minimum 50px on the bottom, which leaves 590px (if we added group headings for each section, we’d subtract another 40px). So a square group on the baseline display would be 590px wide (we’d set the actual height to 100% of its containing grid cell). The total width of the section will then be $(590 * 4 \text{ full-size sections}) + (295 * 1 \text{ half-width section}) + (80 * 4 \text{ for the separator gaps})$. This equals 2975px. To this we’ll add border columns of 120px on the left (according to the silhouette) and 180px on the right, for a total of .

To create a section with exactly this layout, we can use a CSS grid within a block element. To demonstrate this, run Blend and create a new project with the Nav App template (so we just get a basic

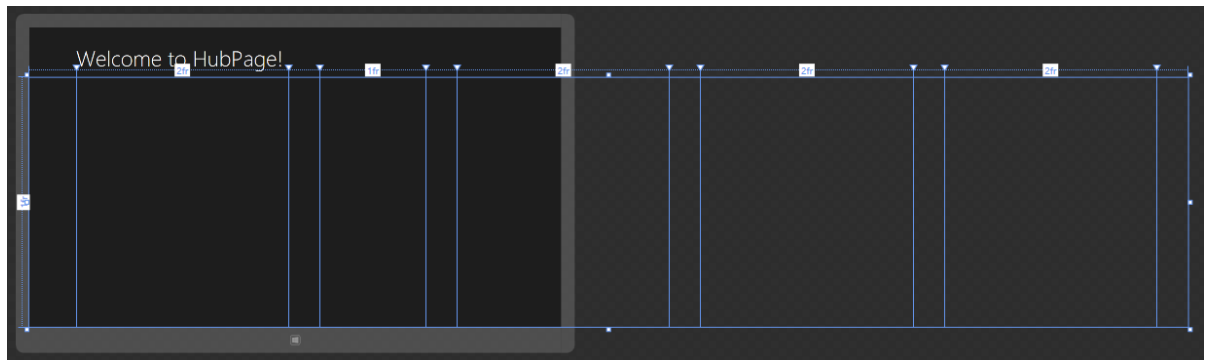
page with the silhouette and not all the secondary pages). Within the the `section` element of `pages/home/home.html`, we create another `div` element and give it a class of *hubSections*:

```
<section aria-label="Main content" role="main">
  <div class="hubSections">
  </div>
</section>
```

Then in `home.css` we add the following style rules. We give `overflow-x: auto` to the `section` element, then lay out our grid in the *hubSections* `div`, using added columns on the left and right for spacing (removing the `margin-left: 120px` from the `section` and adding it as the first column in the `div`):

```
.homepage section[role=main] {
  overflow-x: auto;
}
.homepage .hubSections {
  width: 2975px;
  height: 100%;
  display: -ms-grid;
  -ms-grid-rows: 1fr 50px;
  -ms-grid-columns: 120px 2fr 80px 1fr 80px 2fr 80px 2fr 80px 2fr 80px;
}
```

With just these styles we can already see the hub page taking shape in Blend by zooming out in the artboard:



Now let's create the individual sections, each one starting as a `div` that we add in `home.html`:

```
<section aria-label="Main content" role="main">
  <div class="hubSections">
    <div class="hubSection1"></div>
    <div class="hubSection2"></div>
    <div class="hubSection3"></div>
    <div class="hubSection4"></div>
    <div class="hubSection5"></div>
  </div>
</section>
```

and style them into their appropriate grid cells with 100% `width` and `height`. I'm showing *hubSection1* here as the others are the same with just a different column number (4, 6, 8, and 10, respectively):

```
.homepage .hubSection1 {  
    -ms-grid-row: 1;  
    -ms-grid-column: 2; /* 4 for hubSection2, 6 for hubSection3, etc. */  
    width: 100%;  
    height: 100%;  
}
```

All of this is implemented in the HubPage example included with this chapter.

Laying Out the Sections

Now we can look at the contents of each section. Depending on what you want to display and how you want those sections to interact, you can again just use layout (CSS grids or perhaps flexbox) or use a control like `ListView`. *hubSection3* and *hubSection5* have gaps at the end, so they might be `ListView` controls with variable items. Note that if we created lists with more than 9 or 6 items, respectively, we'd want to adjust the column size in the overall grid and make the `section` element width larger, but let's assume the design calls for a maximum of 9 and 6 items in those sections.

Let's say that we want each section to be interactive, where tapping an item would navigate to a details page. (Not shown in this example are group headers to navigate to a group page.) We'll just then use a `ListView` in each, where each `ListView` has a separate data source. For *hubSection1* we'll need to use cell spanning, but the rest of the groups can just use regular template items. The key consideration with all of these is to style the items so that they fit nicely into the basic dimensions we're using. And referring again back to the silhouette, the spacing between image items should be 10px and the spacing between columns of mixed content (*hubSection4* and *hubSection5*) should be 40px (which can be set with appropriate CSS margins).

Snap Points

If you run the HubPage example and pan around a bit using inertial touch gestures (that is, those that continue panning after you've released your finger, explained more in Chapter 9, "Input and Sensors"), you'll notice that panning can stop in any position along the way. You or your designers might like this, but it also makes sense in many scenarios to automatically stop on a section or group boundary. This can be accomplished for touch interactions using CSS styles for *snap points* as described in the following table. Documentation for these (and some others) can be found on CSS reference for [Touch: Zooming and Panning](#).

Style	Description	Syntax
<code>-ms-scroll-snap-points-x</code>	Defines snap points along the x-axis	<code>snapInterval(start<length>, step<length>)</code> <code>snapList(list<lengths>)</code>

<code>-ms-scroll-snap-points-y</code>	Defines snap points along the y-axis	<code>snapInterval(start<length>, step<length>)</code> <code>snapList(list<lengths>)</code>
<code>-ms-scroll-snap-x</code>	Shorthand to combine <code>-ms-scroll-snap-type</code> and <code>-ms-scroll-snap-points-x</code>	<code>-ms-scroll-snap-type</code> <code>-ms-scroll-snap-points-x</code>
<code>-ms-scroll-snap-y</code>	Shorthand to combine <code>-ms-scroll-snap-type</code> and <code>-ms-scroll-snap-points-y</code>	<code>-ms-scroll-snap-type</code> <code>-ms-scroll-snap-points-y</code>
<code>-ms-scroll-snap-type</code>	Defines what type of snap points should be used for the element: <code>none</code> turns off snap points, <code>mandatory</code> always adjusts panning to land on a snap-point (which includes ending inertial panning), and <code>proximity</code> changes the panning only if a panning motion naturally ends “close enough” to a snap point. Using <code>mandatory</code> , then, will enforce a one-section/item-at-a-time panning behavior, whereas <code>proximity</code> would pan past interim snap points if enough inertia is applied. Note also that dragging with a finger (not using an inertia gesture) will allow the user to pan directly past snap points.	<code>none</code> <code>proximity</code> <code>mandatory</code>

In the table, `<length>` is a floating-point number, followed by an absolute units designator (`cm`, `mm`, `in`, `pt`, or `pc`) or a relative units designator (`em`, `ex`, or `px`).

To add snap points for each of our hub sections, then, we only need to add these styles:

```
.homepage section[role=main] {
  -ms-scroll-snap-type: mandatory;
  -ms-scroll-snap-points-x: snapList(590px, 965px, 1635px, 2305px, 2975px);
}
```

Now you’ll find that panning around stops nicely (with animations) on the section boundaries. Do note that for a hub page like this, proximity snapping is usually more appropriate. Mandatory snap points are intended more for items that can’t be interacted with or consumed without seeing their entirety, such as flipping between pictures, articles, and so on.

For more on this topic, including some of the other `-ms-scroll-*` and `-ms-content-zoom-*` styles, such as scroll rails, refer to the [Input: Pan/scroll and zoom sample](#) in the Windows SDK. Do note also that snap points are not presently supported on the `ListView` control.

Also be clear that snap points are a touch-only feature; if you want to provide the same kind of behavior with mouse and/or keyboard input, you’ll need to do such work manually along the lines of how the `FlipView` control handles transition between items.

The Many Faces of Your Display



If there’s one certainty about layout for a WinRT app, it’s that its display space will likely change over



the lifetime of an app and change frequently. For one, auto-rotation—especially on tablet and slate devices—makes it very quick and simple to switch between landscape and portrait orientations (unlike having to configure a display driver). Second, a device may be connected to an external display, meaning that apps need to adjust themselves to different resolutions on the fly and possibly also different pixel densities. Third, users have the ability in landscape mode to “snap” apps to the left or right side of the screen, where the snapped app is shown in a 320px wide area and another in the “filled” area that occupies the remainder of the display. This is accomplished using touch or mouse gestures, or using the Windows+. (period) and Windows+> (shift+period) keystrokes. (Snapped view requires a display that’s at least 1366x768; otherwise it’s disabled.)

You definitely want to test your app with all of these variances: view states, display sizes, and pixel densities. View states can be tested directly on any given machine, but for the latter two, the Visual Studio simulator and the Device tab of Blend let you simulate different conditions. Our question now is how an app handles them.

View States

We already got an introduction to the four view states in Chapter 1, “The Life Story of a WinRT app,” specifically with Figure 1-6. Let’s now add the next level of precision as described in the following table, which includes an image of the space occupied by the app, a description of the view state, and the identifiers for that state as found in both WinRT (in the [Windows.UI.ViewManagement.ApplicationViewState enumeration](#)) and the [-ms-view-state media feature](#) for CSS media queries:

Space Occupied by the App (Blue)	Details
	<p>App is in landscape mode occupying the entire screen.</p> <p>WinRT: <code>fullScreenLandscape</code></p> <p>-ms-view-state: <code>fullscreen-landscape</code></p>
	<p>App is occupying either left or right side of a landscape screen, in an area that is always 320 pixels wide. This means you do <i>not</i> need to design for all possible sizes between snapped, filled (see below), and full-screen states.</p> <p>WinRT: <code>snapped</code></p> <p>-ms-view-state: <code>snapped</code></p>

	<p>WinRT: <code>filled</code></p> <p>-ms-view-state: <code>filled</code></p> <p>App is occupying the area of the screen next to a snapped app. The width will be the screen size minus 320px minus 22px for the splitter.</p>
	<p>WinRT: <code>fullScreenPortrait</code></p> <p>-ms-view-state: <code>fullscreen-portrait</code></p> <p>App is in portrait mode</p>

Remember again that *every page of your app needs to be prepared for all four view states* (with some exceptions as described in the sidebar below, “Preferred Orientation and Locking Orientation”). View states are always under the user’s control, so any page can be placed into any view state at any time, even on startup (see note below). Repeat this like a mantra, because many designers and developers forget this fact!

Note It’s possible that your app might be launched directly into snap view, as through a user gesture that pulls the app from the left edge of the screen to a snap state. So be prepared for this possibility. Remember also that any extended splash screen in your app is a page that is also subject to view states. In fact, it’s highly likely that a user will snap an app that’s taking a while to load! At the same time, you cannot programmatically control your app’s view state on activation, so it never needs to be saved or restored as part of session state.

An app’s design should thus include all view states for each page, just like we did with the Here My Am! wireframes in Chapter 2. At the same time, handling view states for every page this does not mean four distinct *implementations* of the app. View states are just that: they are different views of the same page content as described on [Guidelines for snapped and fill views](#). That is, switching between view states *always* maintains the state of the app and the page—it *never* changes modes or navigates to another page. The only exception to this rule is that if an app can’t reasonably operate in snap state (like a game that needs a certain amount of screen space to be playable), it can display a message to that effect along with instructions to “Tap here to resume,” which reflects the user’s goal in such a gesture. In response to such a tap, the app can call [Windows.-](#)

[UI.ViewManagement.ApplicationView.tryUnsnap](#), as demonstrated in the [Snap sample](#).³⁴ Don't use this as an excuse to cut corners, however; try as much as possible to keep the app functional in the snapped state.

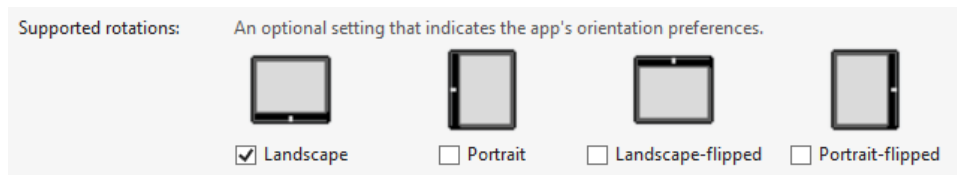
On the flip side, some apps should think about what to do with extra vertical space. A widescreen video in the snapped state will occupy only a small portion of that space, leaving room for, say, additional information about the video, recommendations, playlists, and so on, that wouldn't normally be available when running full screen. In this way, users will find added value in switching to the snapped state.

Sidebar: Preferred Orientation and Locking Orientation

View states aside, it's appropriate for some apps to start in a specific orientation and/or to lock the orientation, effectively ignoring portrait/landscape changes. A movie player, for instance, will generally want to stay in landscape mode, meaning that the fullscreen-landscape and fullscreen-portrait modes are identical—then you can watch videos while laying sideways with a tablet propped up on a chair!

To be clear, the app must still honor the three landscape view states: fullscreen-landscape, filled, and snapped. Preferred orientation is specifically about portrait vs. landscape, and this affects the orientation of your splash screen and other pages in your app. It also enables automatic orientation switching when you switch between your app and others that don't have the same preference.

To tell Windows about your preferences, check the appropriate Supported Orientation boxes in the Application UI tab of the manifest designer:



The many details about how all this works are found on the [InitialRotationPreference page](#) in the documentation. It will also tell you about the [Windows.Graphics.Display.DisplayProperties.autoRotationPreferences](#) and [currentOrientation](#) properties to programmatically control orientation behaviors. For demonstrations, refer to the [Device auto rotation preferences sample](#) in the Windows SDK.

³⁴ [tryUnsnap](#) is the only programmatic API that can affect view states. View states are otherwise always user-initiated, and there are no APIs to set a view state and no way to specify a view state on startup.

Handling View States

As I just mentioned, handling the different view states doesn't mean changing the mode of an app nor does it mean reimplementing a page. Generally speaking, you should try to have feature parity across the states, but in cases like snapped view, especially, the reduced screen real estate will necessitate simplifying the content.

It's best to think about view states simply in terms of the visibility of elements, the size of elements, and their layout on the page. In this way, most of what you need to do can be achieved through CSS media queries using the `-ms-view-state` feature. We saw this again in the Here My Am! app of Chapter 2. The Grid App project template also demonstrates this. Here's how those media queries appear in CSS:

```
@media screen and (-ms-view-state: fullscreen-landscape) {
    /* ... */
}

@media screen and (-ms-view-state: filled) {
    /* ... */
}

@media screen and (-ms-view-state: snapped) {
    /* ... */
}

@media screen and (-ms-view-state: fullscreen-landscape) {
    /* ... */
}

/* Syntax for combining media queries (comma-separated) */
@media screen and (-ms-view-state: fullscreen-landscape),
screen and (-ms-view-state: fullscreen-landscape), screen and (-ms-view-state: filled) {
    /* ... */
}
```

It's also perfectly reasonable to add other clauses to these queries, such as `and (min-width: "1600px")`, as you might be making various other adjustments based on screen sizes.

For WinRT apps, use the view state features in media queries instead of the CSS `orientation` states (landscape and portrait), which are simply derived from the relative width and height of the display and don't distinguish states like `snapped`. In other words, the Windows view states are more specific to the platform and reflect states that the standard CSS does not, helping your app understand not only its available real estate but also the mode in which it's running.³⁵

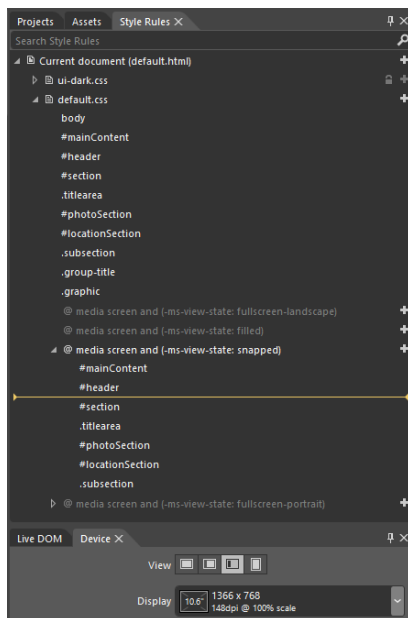
For example, according to the standard CSS algorithm, both the `fullscreen-landscape` and

³⁵ That said, view states are *not* reported to pages loaded into a web context `iframe`. Such pages can use the standard CSS media queries to infer the view state, or the surrounding local context page can pass the view state to the `iframe` through `postMessage`.

snapped states will appear as `orientation: portrait` because the aspect ratio is more vertical than horizontal. However, snapped view implies a different user intent than `fullscreen-landscape`: in snapped view you want to show the most essential parts of an app rather than trying to replicate your portrait layout in a 320-pixels-wide space.

The general practice is to place all your full-screen landscape rules at the top of your CSS file and then make specific adjustments within the specific media queries. We did exactly this with Here My Am! in Chapter 2, where the default styles worked for `fullscreen-landscape` and `filled` as-is, so we needed specific rules only for `snapped` and `fullscreen-landscape`.

Tip When styling your app in Blend, there's a visual affordance in the Style Rules pane that lets you control the exact insertion point of any new CSS styles in the given stylesheet. With this—the orange line shown in the graphic below and shown in Video 2-1 of the companion content—you can indicate where to insert styles for specific media queries and within that media query:



In a few cases, handling media queries in CSS alone won't be sufficient. When the primary content display on a page is a horizontally panning `ListView` with `GridLayout`, you typically switch that control over to `ListLayout` in snapped view. You might also, as suggested on [Guidelines for snapped and fill views](#), change a list of buttons to a single drop-down `select` element to offer the same functionality through a more compact UI.

For these purposes you can employ the standard Media Query Listener API in JavaScript. This interface (part of the W3C CSSOM View Module, see <http://dev.w3.org/csswg/cssom-view/>) allows you to add handlers for media query state changes. To listen for the snapped state, for instance, you can use code like this:

```
var mq1 = window.matchMedia("(-ms-view-state: snapped)");
```

```

mql.addListener(styleForSnapped);

function styleForSnapped() {
    if (mql.matches) {
        //...
    }
}

// Set up listeners for other view states: full-screen, fill, and device-portrait
// or send all media queries to the same handler and check the current state therein.

```

You can see that the media query strings you pass to `window.matchMedia` are the same as used in CSS directly, and in the handler you can, of course, perform whatever actions you need from JavaScript.

Tip Be sure to test your view states coming out of the suspending state after display characteristics might have changed, such as plugging in a different monitor or going to the Settings charm > Change PC Settings > Ease of Access and toggling Make Everything on the Screen Bigger. That is, it's possible to bring your app from the background (suspended state) directly into snap view, and screen dimensions might also have changed while you're suspended. So test your layout when resuming into snap and when resuming into different screen dimensions. If necessary, add a resuming handler to take any needed steps under these conditions.

When handling view states (or `window.onresize` events), you can obtain exact dimensions of your app window through the `window.innerWidth` and `window.innerHeight` properties. The `document.body.clientWidth` and `document.body.clientHeight` properties will be the same, as will be the `clientWidth` and `clientHeight` properties of any element (like a `div`) that occupies 100% of the document body. Within the `resize` event, the `args.view.outerWidth` and `args.view.outerHeight` properties are also available.

In CSS there are also variables for the viewport height and viewport width: `vh` and `vw`. You can prefix these with a percentage number, such that 100vh is 100% of the viewport height, and 3.5vw is 3.5% of the viewport width. These variables can also be used in CSS `calc` expressions.

The current view state is also available through the `Windows.UI.ViewManagement.ApplicationViewState` API. Its return value comes from the `Windows.UI.ViewManagement.ApplicationViewState` enumeration as shown in the earlier table. We've seen a few uses of this in earlier chapters. For instance, page controls (discussed in Chapter 3, "App Anatomy and Page Navigation") typically check the view state within their `ready` method and directly receive those states within their `updateLayout` method. In fact, every method of the `groupedItems` page control in the Grip App project template is sensitive to the view state. Take a look at the `groupedItems.js`:

```

// A few lines and comments are omitted
var appView = Windows.UI.ViewManagement.ApplicationView;
var appViewState = Windows.UI.ViewManagement.ApplicationViewState;
var nav = WinJS.Navigation;
var ui = WinJS.UI;

```

```

ui.Pages.define("/pages/groupedItems/groupedItems.html", {
    initializeLayout: function (listView, viewState) {
        if (viewState === appViewState.snapped) {
            listView.itemDataSource = Data.groups.dataSource;
            listView.groupDataSource = null;
            listView.layout = new ui.ListLayout();
        } else {
            listView.itemDataSource = Data.items.dataSource;
            listView.groupDataSource = Data.groups.dataSource;
            listView.layout = new ui.GridLayout({ groupHeaderPosition: "top" });
        }
    },

    itemInvoked: function (args) {
        if (appView.value === appViewState.snapped) {
            // If the page is snapped, the user invoked a group.
            var group = Data.groups.getAt(args.detail.itemIndex);
            nav.navigate("/pages/groupDetail/groupDetail.html", { groupKey: group.key });
        } else {
            // If the page is not snapped, the user invoked an item.
            var item = Data.items.getAt(args.detail.itemIndex);
            nav.navigate("/pages/itemDetail/itemDetail.html", { item: Data.getItemReference(item) });
        }
    },

    ready: function (element, options) {
        // ...
        this.initializeLayout(listView, appView.value);
        // ...
    },

    // This function updates the page layout in response to viewState changes.
    updateLayout: function (element, viewState, lastViewState) {
        var listView = element.querySelector(".groupeditemslist").winControl;
        if (lastViewState !== viewState) {
            if (lastViewState === appViewState.snapped || viewState === appViewState.snapped) {
                var handler = function (e) {
                    listView.removeEventListener("contentanimating", handler, false);
                    e.preventDefault();
                }
                listView.addEventListener("contentanimating", handler, false);
                this.initializeLayout(listView, viewState);
            }
        }
    }
});

```

First, the `initializeLayout` method that's called from both `ready` and `updateLayout` checks the current view state and adjusts the `ListView` control accordingly. If you remember from Chapter 5, it's perfectly allowable to change a `ListView`'s layout and data source properties on the fly; here we use a `ListLayout` with a list of groups for snapped view and a `GridLayout` with grouped items in all others. This demonstrates how we're showing the same content but in a more concise manner by hiding the individual items in snapped view. Because of this, `itemInvoked` also has to check the view state

because the list items are groups in snapped view and should navigate to a group details page.

As for `updateLayout`, this is invoked from a `window.onresize` event handler in the `PageControlNavigator` code (see `js/navigator.js` in the Grid App project template). That handler passes the new and previous view states to `updateLayout`. If that function detects that we're switching to or from snapped state, it resets the `ListView` through `initializeLayout`. And because we're changing the `ListView`'s data source, there's no need to play entrance or transition animations. The little trick that's played with the `contentanimating` event here simply suppresses those.

Sidebar: Physical Display Orientations

The fullscreen-landscape and fullscreen-portrait view states suggest something of how a device is actually oriented in physical space, but such information is more accurately derived from properties of the [Windows.Graphics.Display.DisplayProperties object](#). Specifically, the `currentOrientation` property contains a value from [Windows.Graphics.Display.DisplayOrientations](#) that indicates how the device is rotated in relation to its `nativeOrientation` (and an `orientationchanged` event fires when needed). This can tell you, for example, whether the device is being held upside-down against the sky, which would be useful for any kind of augmented reality app such as a star chart.

Similarly, the APIs in [Windows.Devices.Sensors](#), specifically the `SimpleOrientationSensor` and `OrientationSensor` classes can provide more information from the hardware itself.

Screen Size, Pixel Density, and Scaling

I don't know about you, but when I first read that the snapped area was *always* 320 pixels—real pixels, not a percentage of the screen width—it really set me wondering. Wouldn't that give a significantly different user experience on different monitors? The answer is actually no. 320 pixels is about 25% of the baseline 1366x768 target display, which means that the remaining 75% of the screen is a familiar 1024x768. And on a 10-inch screen, it means that snap area is about the 2.5 physical inches wide. So far so good.

With a large monitor, on the other hand, let's say a 2560x1440 monster, those 320 pixels would only be 12.5% of the width, so the layout of the whole screen looks quite different. However, given that such monitors are in the 24-inch range, those 320 pixels still end up being about 2.5 physical inches wide, meaning that the snap area gives essentially the same visual experience as before, just now with much more vertical space to play with and much more remaining screen space.

But this now brings up the question of *pixel density*—what happens if your app ends up on a really small screen that also has a really high resolution? Obviously, 320 pixels on the latter display would be little more than an inch wide. Anyone got a magnifying glass?

Fortunately, this isn't anything a WinRT app has to worry about...almost. The main user benefit for such displays is greater sharpness, not greater density of information. Touch targets need to be the

same size on any size display no matter how many pixels it occupies, because human fingers don't change with technology! To accommodate this, Windows automatically scales down the effective resolution that's reported to apps, which is to say that whatever coordinates you use within your app (in HTML, CSS, and JavaScript) get automatically scaled up to the necessary device resolution. This happens at within the low-level HTML/CSS rendering engine in the app host so that everything is drawn directly against native device pixels for maximum sharpness.

As for the "almost" above, the one place where you do need to care about pixel density is with raster graphics, as we discussed in Chapter 3 for your splash screen and tiles. We'll return to this shortly in the "Graphics that Scale Well" section below.

Display sizes and pixel densities can both be tested again using the Visual Studio simulator or the Device tab in Blend. The latter, shown in Figure 6-5, indicates the applicable DPI and scaling factor. 100% scale means the device resolution is reported directly to an app. 140% and 180%, on the other hand, indicate that scaling is taking place. With the 10.6" 2560x1440 setting with 180%, for example, the app will see dimensions of 1422x800 (2560/1.8 by 1440/1.8), which is very close to the standard 1366x768 display; similarly, the 10.6: 1920x1080 setting with 140% scaling will appear to the app as 1371x771 (1920/1.4 by 1080/1.4). In both cases, a layout designed for 1366x768 is completely sufficient though you can certainly be as precise as you want.

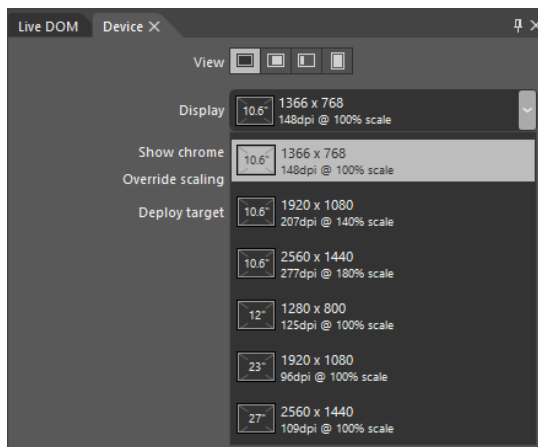


Figure 6-5 Options for display sizes and pixel densities in Blend's Device tab.

As noted earlier with view states, you can programmatically determine the exact size of your app window through the `window.innerWidth` and `window.innerHeight` properties, the `document.body.clientWidth` and `document.body.clientHeight` properties, and the `clientWidth` and `clientHeight` properties of any element that occupies 100% of the body. Within `window.onresize`, you can use these (or the `args.view.outerWidth` and `args.view.outerHeight` properties) to adjust the app's layout for changes in the overall display. Of course, if you're using something like the CSS grid with fractional rows and columns to do your layout, much of that will be handled automatically.

In all cases, these dimensions will already reflect automatic scaling for pixel densities, so they are the dimensions against which you want to determine layout. If you want to know the physical *display* dimensions, on the other hand, you'll find these in the `window.screen.width` and `window.screen.height` properties. Other aspects of the display can be found in the `Windows.Graphics.Display.DisplayProperties` object, such as the `logicalDPI` and the current `resolutionScale`. The latter is a value from the `Windows.Graphics.Display.ResolutionScale enumeration`, one of `scale100Percent`, `scale140Percent`, and `scale180Percent`. The actual values of these identifiers are 100, 140, and 180 so that you can use `resolutionScale` directly in calculations.

Sidebar: A Good Opportunity for Remote Debugging

Working with different device capabilities provides a great opportunity to work with remote debugging as described on [Running Windows WinRT apps on a remote machine](#). This will help you test your app on different displays without needing to set up Visual Studio on each one, and it also gives you the benefit of multimonitor debugging. You only need to install and run the remote debugging tools on the target machine and make sure it's connected with a cable to the same network as your development machine. (You might need to buy a small USB-Ethernet adapter if your device doesn't have a suitable port—remote debugging doesn't work over the Internet, and it doesn't work over wireless networks.) The Remote Debugging Monitor running on the remote machine will announce itself to Visual Studio running on your development machine. Note that the first time you run the app remotely, you'll be prompted to obtain a developer license for that machine, so it will need to be connected to the Internet during that time.

Graphics that Scale Well

Variable screen sizes and pixel densities can present a bit of a challenge to apps, not just in layout but also in making sure that graphical assets always look their best. You can certainly draw graphics directly with the HTML5 `canvas`; what I want to specifically address are predrawn assets.

HTML5 scalable vector graphics (SVGs) are very handy here. You include inline SVGs in your HTML (including page fragments), or you can keep them in separate files and refer to them in an `img.src` attribute. One of the easiest ways to use an SVG is to place an `img` element inside a proportionally sized cell of a CSS grid and set the element's `width` and `height` styles to 100%. The SVG will then automatically scale to fill the cell, and since the cell will resize with its container, everything is handled automatically.

One caveat with this approach is that the SVG will be scaled to the aspect ratio of the containing grid cell, which isn't always what you want. To control this behavior, make sure the SVG has `viewBox` and `preserveAspectRatio` attributes where the `viewBox` aspect ratio matches that defined by the SVG's `width` and `height` properties:

```
<svg
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns="http://www.w3.org/2000/svg"
```



```

xmlns:xlink="http://www.w3.org/1999/xlink"
version="1.0"
width="300"
height="150"
viewBox="0 0 300 150"
preserveAspectRatio="xMidYMid meet">

```

Of course, you don't always have nice vector graphics. Bitmaps that you include in your app package, pictures you load from files, and raster images you obtain from a service won't be so easily scalable. In these cases, you'll need to be aware of and apply the current scaling factor appropriately.

For assets in your app package, we already saw how to work with varying pixel densities in Chapter 3 through the `.scale-100`, `.scale-140`, and `.scale-180` file name suffixes. These work for any and all graphics in your app, just as they do for the splash screen, tile images, and the other graphics referenced by the manifest. So if you have a raster graphic names `banner.png`, you'll create three graphics in your app package called `banner.scale-100.png`, `banner.scale-140.png`, and `banner.scale-180.png`. You can then just refer to the base name in an element or in CSS, as in `` and `background-image: url('images/banner.png')`, and the Windows resource loader will magically load the appropriately scaled graphic automatically. (If files with `.scale-*` suffixes aren't found, it will look for `banner.png` directly.) We'll see even more such magic in Chapter 17, "Apps for Everyone," when we also include variants for different languages and contrast settings that introduce additional suffixes of their own.

If your sensibilities as a developer object to this file-naming scheme, know that you can also use similarly named folders instead. That is, create `scale-100`, `scale-140`, and `scale-180` folders in your `images` folder and place appropriate files with unadorned names (like `banner.png`) therein.

In CSS you can also use media queries with `max-resolution` and `min-resolution` settings to control which images get loaded. Remember, however, that CSS will see the logical DPI, not the physical DPI, so the cutoffs for each scaling factor are as follows:

```

@media all and (max-resolution: 134dpi) {
    /* 100% scaling */
}

@media all and (min-resolution: 135dpi) {
    /* 140% scaling */
}

@media all and (min-resolution: 174dpi) {
    /* 180% scaling */
}

```

As explained in the [Guidelines for scaling to pixel density](#), such media queries are especially useful for images you obtain from a remote source, where you might need to change the specific URI or the URI query string.

Programmatically, you can again obtain `logicalDpi` and `resolutionScale` properties from the `Windows.Graphics.Display.DisplayProperties` object. Its `logicalDpiChanged` event can also be

used to check for changes in the `resolutionScale`, since the two are always coupled. Usage of these APIs is demonstrated in the [Scaling according to DPI sample](#).

If your app manages a cache of graphical assets, by the way, especially those downloaded from a service, include the `resolutionScale` value for which that graphic was obtained. This way you can obtain a better image if and when necessary, or you can scale down a higher resolution image that you already obtained. It's also something to be aware of with any app settings you might roam, because the pixel density and screen size may vary between a user's devices.

Adaptive and Fixed Layouts for Display Size

Just as every page of your app needs to be prepared for different view states, it should also be prepared for different screen sizes. On this subject, I recommend you read the [Guidelines for scaling to screens](#), which has good information on the kinds of display sizes your app might encounter. From this we can conclude that the smallest snapped view you'll ever encounter is 320x768, the minimum filled view is 1024x768, and the minimum full-screen views (portrait and landscape) are 1280x800 and 1366x768. These are your basic design targets.

From there, displays only get larger, so the question becomes "What do you do with more space?" The first part of the answer is "Fill the screen!" Nothing looks sillier than an app running on a 27" monitor that was designed and implemented with only 1366x768 in mind, because it will only occupy a quarter to half of the screen at best. As I've said a number of times, imagine the kinds of reviews and ratings your app might be given in the Windows Store if you don't pay attention to certain details!

The second part of the answer depends on your app's content. If you have only fixed content, which is common with games, then you'll want to use a fixed layout that scales to fit. If you have variable content, meaning that you should show more when there's more screen space, then you want to use an adaptive layout. Let's look at both of these in turn.

Sidebar: The Make Everything on Your Screen Bigger Setting

In the PC Settings app (invoke the Settings charm and select Change PC Settings in the lower-right corner), there is an option within Ease of Access to "Make everything on your screen bigger." Turning this on effectively enlarges the display by about 40%, meaning that the system will report a screen size to the app that's about 30% smaller than the current scaled resolution (similar to the 140% scaling level). Fortunately, this setting is disabled if it would mean reporting a size smaller than 1024x768, which always remains the minimum screen size your app will encounter. In any case, when this setting is changed it will trigger a `Windows.Graphics.Display.DisplayProperties.logicalDpiChanged` event.

Fixed Layouts and the ViewBox Control

A fixed layout is the best choice for apps that aren't oriented around variable content, because there isn't more content to show on a larger screen. Such an app instead need to scale its output to fill the display as best it can, depending on whether it needs to maintain an aspect ratio.

An app can certainly obtain the dimensions of its display window and redraw itself accordingly. Every coordinate in the app would be a variable in this case, and elements would be resized and laid out relative to one another. Such an approach is great when an app can adapt its aspect ratio to that of the screen, thereby filling 100% of the display.

You can do the same thing with a fixed aspect ratio as well by placing limits on your coordinates, perhaps by using an absolute coordinate system to which you then apply your own scaling factor.

Because this is the more common approach, WinJS provides a built-in layout control for exactly this purpose: `WinJS.UI.ViewBox`. Like all other WinJS controls, you can declare this using `data-win-control` in HTML as follows, where the ViewBox element can contain one and only one child element:

```
<div data-win-control="WinJS.UI.ViewBox">
  <div class="fixedlayout">
    <p>Content goes here</p>
  </div>
</div>
```

This is really all you ever see with the ViewBox as it has no other options or properties, no methods, and no events—very simple! Note also that because the ViewBox is just a control, you can use it for any fixed aspect-ratio content in an otherwise adaptive layout; it's not only for the layout of an entire page.

To set the reference size of the ViewBox—the dimensions against which you'll write the rest of your code—simply set the `width` and `height` styles of the child element in CSS. For example, to set a base size of 1024x768, we'd set those properties in the rule for the `fixedlayout` class:

```
.fixedlayout {
  width: 1024px;
  height: 768px;
}
```

Once instantiated, the ViewBox simply listens for `window.onresize` events, and it then applies a CSS 2D scaling transform to its child element based on the difference between the reference size and the actual size, preserving the aspect ratio. This works to scale the contents up as well as down. Automatic letterboxing or sidepillars are also applied around the child element, and you can set the appearance of those areas (really any area not obscured by the child element) by using the `win-viewbox` class. As always, scope that selector to your specific control if you're using more than one ViewBox in your app, unless you want styles to be applied everywhere.

The basic structure above is what you get with a new app created from the Fixed Layout App project template in Visual Studio and Blend. As shown here, it creates a layout with a 1024x768

reference size, but you can use whatever dimensions you like.

The CSS for this project template reveals that the whole page itself is actually styled as a CSS flexbox to make sure the ViewBox is centered, and that the `fixedlayout` element is given a default grid:

```
html, body {
    height: 100%;
    margin: 0;
    padding: 0;
}

body {
    -ms-flex-align: center;
    -ms-flex-direction: column;
    -ms-flex-pack: center;
    display: -ms-flexbox;
}

.fixedlayout {
    -ms-grid-columns: 1fr;
    -ms-grid-rows: 1fr;
    display: -ms-grid;
    height: 768px;
    width: 1024px;
}
```

If you create a project with this template in Blend, add a border style to `fixedlayout` (like `border: 2px solid Red;`), and fiddle with the view states and the display settings on the Device tab, you can see how the ViewBox provides all the scaling for free. To show this more obviously, the FixedLayout example for this chapter changes the child element of the ViewBox to a `canvas` on which it draws a 4x3 grid (to match the aspect ratio of 1024x768) of 256px squares containing circles. As shown in Figure 6-6, the squares and circles don't turn into rectangles and ovals as we move between view states and display sizes, and letterboxing is handled automatically (applying a `background-color` style to the `win-viewbox` class).

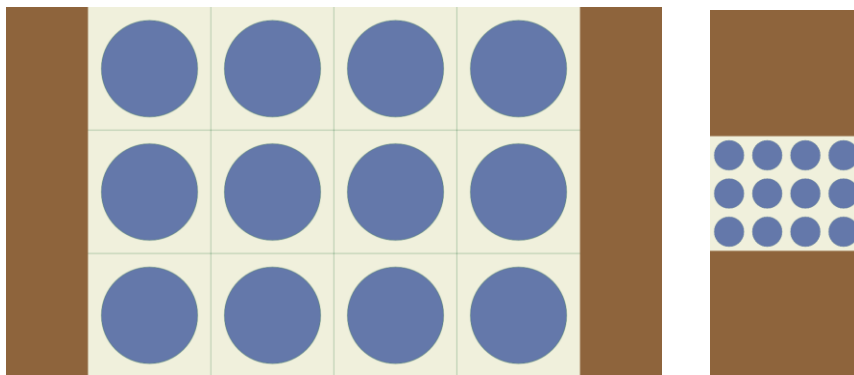


Figure 6-6 Fixed layout scaling with the WinJS.UI.ViewBox controls, showing letterboxing on a full-screen 1366x768 display (left) and in snap view (right).

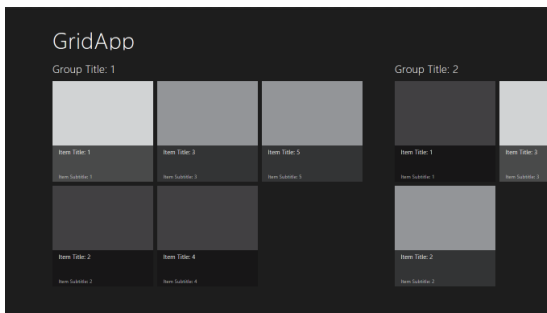
Sidebar: Raster Graphics and Fixed Layouts

If you use raster graphics within a ViewBox, size them according to the maximum 2560x1440 resolution so that they'll look good on the largest screens and they'll still scale down to smaller ones (rather than being stretched up). Alternately, you can use load different graphics (through different `img.src` URLs) that are better suited for the most common screen size.

Note that resolution scaling will still be applicable. If you're running on a high-density 10.6" 2560x1440 display (180% scale), the app and thus the ViewBox will still see smaller screen dimensions. But if you're supplying a graphic for the native device resolution, it will look sharp when rendered on the screen.

Adaptive Layouts

Adaptive layouts are those in which an app shows more content when more screen space is available. Such a layout is most easily achieved with a CSS grid where proportional rows and columns will automatically scale up and down; elements within grid cells will then find themselves resized accordingly. This is demonstrated in the Visual Studio/Blend project templates, especially the Grid App project template. On a typical 1366x768 display you'll see a few items on a screen, as shown at the top of Figure 6-7. Switch over to a 27" 2560x1440 and you'll see a lot more, as shown at the bottom.



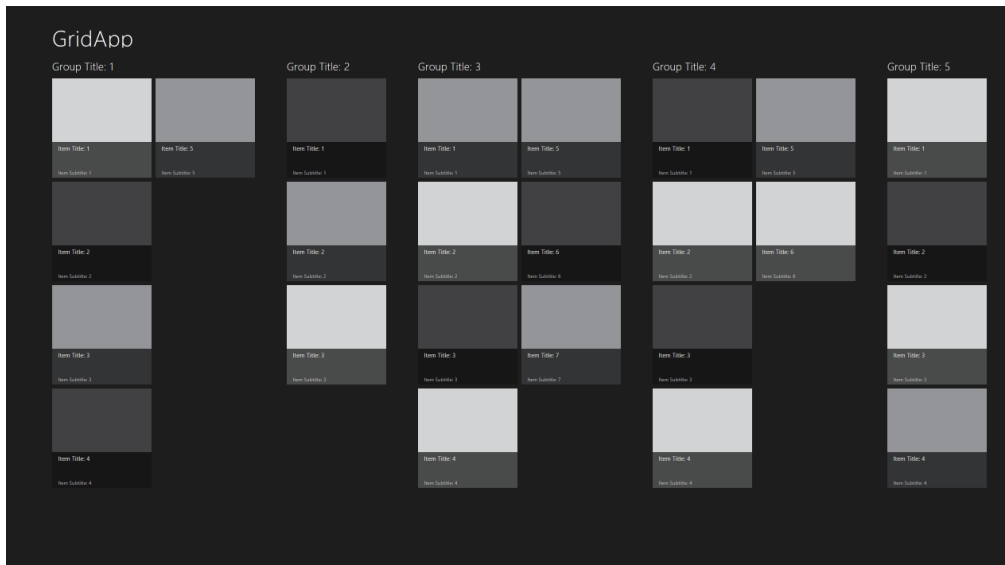


Figure 6-7 Adaptive layout in the Grid App project template shown for a 1366x768 display (top) and a 2560x1440 display (bottom).

To be honest, the Grip App project template doesn't do anything different for display size than it already does for view states. Because it uses CSS grids and proportional cell sizes, the cell containing the `ListView` control automatically becomes bigger. Since the `ListView` control is listening for `window.onresize` on its own, we don't need to separately instruct it to update its layout.

The overall strategy for an adaptive layout, then, is straightforward:

- Use a CSS grid where possible to handle adaptive layout automatically.
- Listen for `window.onresize` as necessary to reposition and resize elements manually, such as an HTML canvas `element`.
- Have controls listen to `window.onresize` to adapt themselves directly. This is especially important for collection controls like `ListView`.

As another reference point, refer to the [Adaptive layout with CSS sample](#), which really takes the same approach as the Grid App project template, relying on controls to resize themselves. As you can see, the app isn't doing any direct calculations based on window size.

Hint If you have an adaptive layout and want a background image specified in CSS to scale to its container (rather than being repeated), style `background-size` to be either `contain` or `100% 100%`.

It should be also clear to you as a developer that how an app handles different screen sizes is also a design matter. The strategy above is what you use to implement a design, but the design still needs to

think about how everything should look. The following considerations, which I only summarize here, are described on [Guidelines for scaling to screens](#):

- Which regions are fixed and which are adaptive?
- How do adaptive regions makes use of available space, including the directions in which that region adapts?
- How do adaptive and fixed regions relate in the wireframe?
- How does the app’s layout overall makes use of space—that is, how does whitespace itself expand so that content doesn’t become too dense?
- How does the app make use of multicolumn text?

Answering these sorts of questions will very much help you understand how the layout should adapt.

Using the CSS Grid

Starting back in Chapter 2, we’ve already been employing CSS grids for many purposes. Personally, I love the grid model because it so effortlessly allows for relative placement of elements that automatically scales to different screen sizes.

Because the focus of this book is on the specifics of Windows 8, I’ll leave it to the W3C specs on <http://www.w3.org/TR/css3-grid-layout/> and <http://dev.w3.org/csswg/css3-grid-align/> to explain all the details. These specs are essential references for understanding how rows and columns are sized, especially when some are declared with fixed sizes, some are sized to content, and others are declared such that they fill the remaining space. The nuances are many!

Because the specs themselves are still in the “Editor’s Draft” stage as of this writing, it’s good to know exactly which parts of those specs are actually supported by the HTML/CSS engine used for WinRT apps.

For the element containing the grid, the supported styles are simple:

- `-ms-grid` and `-ms-inline-grid` as display models (the `display` style). We’ll come back to `-ms-inline-grid` later.
- `-ms-grid-columns` and `-ms-grid-rows` on the grid element, to define its arrangement. If left unspecified, the default is one column and one row. The repeat syntax such as `-ms-grid-columns: (1fr)[3];` is supported, which is most useful when you have repeated series of rows or columns, which appear inside the parentheses. As examples, all the following are equivalent:

```
-ms-grid-rows:10px 10px 10px 20px 10px 20px 10px;  
-ms-grid-rows:(10px)[3] (20px 10px)[2];  
-ms-grid-rows:(10px)[3] (20px 10px) 20px 10px;  
-ms-grid-rows:(10px)[2] (10px 20px)[2] 10px;
```

How you define your rows and columns is the really interesting part, because you can make some fixed, some flexible, and some sized to the content using the following values. Again, see the specs for the nuances involving `max-content`, `min-content`, `minmax`, `auto`, and `fit-content` specifiers, along with values specified in units of `px`, `em`, `%`, and `fr`. WinRT apps can also use `vh` (viewport height) and `vw` (viewport width) as units.

Within the grid now, child elements are placed in specific rows and columns, with specific alignment, spanning, and layering characteristics using the following styles:

- `-ms-grid-column`: identifies the 1-based column of the child in the grid.
- `-ms-grid-row`: identifies the 1-based row of the child in the grid.
- `-ms-grid-column-align` and `-ms-grid-row-align` specify where the child is placed in the grid cell. Allowed values are `start`, `end`, `center`, and `stretch` (default).
- `-ms-grid-column-span` and `-ms-grid-row-span` indicate that a child spans one or more rows/columns.
- `-ms-grid-layer` controls how grid items overlap. This is similar to the `z-index` style as used for positional element. Since grid children are not positioned directly with CSS and are instead positioned according to the grid, `-ms-grid-layer` allows for separate control.

Be very aware that row and column styles are 1-based, not 0-based. Really re-program your JavaScript-oriented mind to remember this, as you'll need to do a little translation if you track child elements in a 0-based array.

Also, when referring to any of these `-ms-grid*` styles as properties in JavaScript, drop the hyphens and switch to camel case, as in `msGrid`, `msGridColumn`, `msGridRowAlign`, `msGridLayer`, and so on.

Overall, grids are fairly straightforward to work with, especially within Blend where you can immediately see how the grid is taking shape. Let's now take a look at a few tips and tricks that you might find useful.

Overflowing a Grid Cell

One of the great features of the grid, depending on your point of view, is that overflowing content in a grid cell doesn't break the layout at all—it just overflows. (This is very different from tables!) What this means is that you can, if necessary, offset a child element within a grid cell so that it overlaps an adjacent cell (or cells). Besides not breaking the layout, this makes it possible to animate elements moving between cells in the grid, if desired.

A quick example of content that extends outside its containing grid cell can be found in the `GridOverflow` example with this chapter's companion content. For the most part, it creates a 4x4 grid of rectangles, but this code at the end of the `doLayout` function (`default.js`), places the first rectangle well outside its cell:


```
children[0].style.width = "350px";
children[0].style.marginLeft = "150px";
children[0].style.background = "#fbb";
```

This makes the first element in the grid wider and moves it to the right, thereby making it appear inside the second element's cell (the background is changed to make this obvious). Yet the overall layout of the grid remains untouched.

Why I cast a little doubt on this being a great feature is that you might not want this behavior at times, hoping instead that the grid would resize to the content. For that behavior, try using an HTML table.

Centering Content Vertically

Somewhere in your own experience with CSS, you've probably made the bittersweet acquaintance with the `vertical-align` style in an attempt to place a piece of text in the middle of a `div`, or at the bottom. Unfortunately, it doesn't work: this particular style works only for table cells and for inline content (to determine how text and images, for instance, are aligned in that flow).

As a result, various methods have been developed to do this, such as those discussed in <http://blog.themeforest.net/tutorials/vertical-centering-with-css/>. Unfortunately, just about every technique depends on fixed heights—something that can work for a website but doesn't work well for the adaptive layout needs of a WinRT app. And the one method that doesn't use fixed heights uses an embedded table. Urk.

Fortunately, both the CSS grid and the flexbox (see "Item Layout" later on) easily solve this problem. With the grid, you can just create a parent `div` with a 1x1 grid and use the `-ms-grid-row-align: center` style for a child `div` (which defaults to cell 1, 1):

```
<!-- In HTML -->
<div id="divMain">
  <div id="divChild">
    <p>Centered Text</p>
  </div>
</div>

/* In CSS */
#divMain {
  width: 100%;
  height: 100%;
  display: -ms-grid;
  -ms-grid-rows: 1fr;
  -ms-grid-columns: 1fr;
}

#divChild {
  -ms-grid-row-align: center;
  -ms-grid-column-align: center;

  /* Horizontal alignment of text also work with the following */
```

```

    /* text-align: center; */
}

```

The solution is even simpler with the `flexbox` layout, where `flex-align: center` handles vertical centering, `flex-pack: center` handles the horizontal, and a child `div` isn't needed at all. This is the same styling that's used in the Fixed Layout App project template to center the `ViewBox`:

```

<!-- In HTML -->
<div id="divMain">
    <p>Centered Text</p>
</div>

/* In CSS */
#divMain {
    width: 100%;
    height: 100%;
    display: -ms-flexbox;
    -ms-flex-align: center;
    -ms-flex-direction: column;
    -ms-flex-pack: center;
}

```

Code for both these methods can be found in the `CenteredText` example for this chapter. (This example is also used to demonstrate the use of ellipsis later on, so it's not exactly as it appears above.)

Scaling Font Size

One particularly troublesome area with HTML is figuring out how to scale a font size with an adaptive layout. I'm not suggesting you do this with the standard typography recommended by WinRT app design as we saw earlier in this chapter—it's more a consideration when you need to use fonts in some other aspect of your app such as large letters on a tile in a game.

With an adaptive layout, you typically want certain font sizes to be proportional to the dimensions of its parent element. (It's not a concern if the parent element is a fixed size, because then you can fix the size of the font.) Unfortunately, percentage values used in the `font-size` style in CSS are based on the default font size (1em), not the size of the parent element as happens with `height` and `width`. What you'd love to be able to do is something like `font-size: calc(height * .4)`, but, well, the value of other CSS styles on the same element are just not available to `calc`.

One exception to this is the `vh` value (which can be used with `calc`). If you know, for instance, that the text you want to scale is contained within a grid cell that is always going to be 10% of the viewport height, and if you want the font size to be half of that, then you can just use `font-size: 5vh` (5% of viewport height).

Another method is to use an SVG for the text, wherein you can set a `viewbox` attribute and a `font-size` relative to that `viewbox`. Then scaling the SVG to a grid cell will effectively scale the font:

```

<svg viewBox="0 0 600 400" preserveAspectRatio="xMaxYMax">
    <text x="0" y="150" font-size="200" font-family="Verdana">
        Big SVG Text
    </text>
</svg>

```

```
</text>
</svg>
```

You can also use JavaScript to calculate the desired font size programmatically based on the `clientHeight` property of the parent element. If that element is in a grid cell, the font size (and line height) can be some percentage of that cell's height, thereby allowing the font to scale with the cell.

You can also try using the `WinJS.UI.ViewBox` control. If you want content like text to take up 50% of the containing element, wrap the ViewBox in a `div` that is styled to be 50% of the container and style the child element of the ViewBox with `position: absolute`. Try dropping the following code into `default.html` of a new Blank app project for a demonstration:

```
<div style="height:50%;">
  <div data-win-control="WinJS.UI.ViewBox">
    <p style="position:absolute;">Big text!</p>
  </div>
</div>
```

Item Layout

So far in this chapter we've explored page-level layout, which is to say, how top-level items are positioned on a page, typically with a CSS grid. Of course, it's all just HTML and CSS, so you can use tables, line breaks, and anything else supported by the rendering engine so long as you adapt well to view states and display sizes.

It's also important to work with item layout in the flexible areas of your page. That is, if you set up a top-level grid to have a number of fixed-size areas (for headings, title graphics, control bars, etc.), the remaining area can vary greatly in size as the window size changes. In this section, then, let's look at some of the tools we have to within those specific regions: CSS transforms, flexbox, nested and inline grids, multicolumn text, CSS figures, and CSS connected frames. A general reference for these and all other CSS styles that are supported for WinRT apps (such as background, borders, and gradients) can be found on the [Cascading Style Sheets](#) topic.

CSS 2D and 3D Transforms

It's really quite impossible to think about layout for elements without taking CSS transforms into consideration. Transforms are very powerful because they make it possible to change the display of an element without actually affecting the document flow or the overall layout. This is very useful for animations and transitions; transforms are used heavily in the WinJS animations library that provides the Windows 8 look and feel for all the built-in controls. As we'll explore in Chapter 11, "Purposeful Animations," you can make direct use of this library as well.

CSS transforms can also be used directly, of course, anytime you need to translate, scale, or rotate an element. Both 2D and 3D transforms (<http://dev.w3.org/csswg/css3-2d-transforms/> and

<http://www.w3.org/TR/css3-3d-transforms/>) are supported for WinRT apps, specifically these styles:³⁶

CSS Style	JavaScript Property (element.style.)
<code>backface-visibility</code>	<code>backfaceVisibility</code>
<code>perspective</code> , <code>perspective-origin</code>	<code>perspective</code> , <code>perspectiveOrigin</code>
<code>transform</code> , <code>transform-origin</code> , and <code>transform-style</code>	<code>transform</code> , <code>transformOrigin</code> , and <code>transformStyle</code>

Full details can be found on the [Transforms](#) reference. Know also that because the app host uses the same underlying engines as Internet Explorer, transforms enjoy all the performance benefits of hardware acceleration.

Flexbox

Just as the grid is magnificent for solving many long-standing problems with page layout, the CSS flexbox module, documented at <http://www.w3.org/TR/css3-flexbox/>, is excellent for handling variable-sized areas wherein the content wants to “flex” with the available space. To quote the W3C specification:

In this new box model, the children of a box are laid out either horizontally or vertically, and unused space can be assigned to a particular child or distributed among the children by assignment of ‘flex’ to the children that should expand. Nesting of these boxes (horizontal inside vertical, or vertical inside horizontal) can be used to build layouts in two dimensions.

As the flexbox spec is presently in draft form, the specific display styles for WinRT apps are `display: -ms-flexbox` (block level) and `display: -ms-inline-flexbox` (inline).³⁷ For a complete reference of the other supported properties, see the [Flexible Box \(“Flexbox”\) Layout](#) documentation:

CSS Style	JavaScript Property (element.style.)	Values
<code>-ms-flex-align</code>	<code>msFlexAlign</code>	start end center baseline stretch
<code>-ms-flex-direction</code>	<code>msFlexDirection</code>	row column row-reverse column-reverse inherit
<code>-ms-flex-flow</code>	<code>msFlexFlow</code>	<code><direction> <pack></code> where <code><direction></code> is an <code>-ms-flex-direction</code> value and <code><pack></code> is an

³⁶ At the time of writing, the `-ms-*` prefixes on these styles were dropped but are still supported.

³⁷ If you’re accustomed to the `-ms-box*` styles for flexbox, Microsoft has since aligned to the W3C specifications that are expected to be the last major revision before the standard is finalized. As the new syntax replaces the old, the old will not work in WinRT apps nor Internet Explorer 10.

		-ms-flex-pack value.
-ms-flex-orient	msFlexOrient	horizontal vertical inline-axis block-axis inherit
-ms-flex-item-align	msFlexItemAlign	auto start end center baseline stretch
-ms-flex-line-pack	msFlexLinePack	start end center justify distribute stretch
-ms-flex-order	msFlexOrder	<integer> (ordinal group)
-ms-flex-pack	msFlexPack	start end center justify
-ms-flex-wrap	msFlexWrap	none wrap wrapreverse

As with all styles, Blend is a great tool in which to experiment with different flexbox styles because you can see the effect immediately. It's also helpful to know that flexbox is used in a number of places around WinJS and in the project templates, as we saw with the Fixed Layout template earlier. The ListView control in particular takes advantage of it, allowing more items to appear when there's more space. The FlipView uses flexbox to center its items, and the Ratings, DatePicker, and TimePicker controls all arrange their inner elements using an inline flexbox. It's likely that your own custom controls will do the same.

Nested and Inline Grids

Just as the flexbox has both block level and inline models, there is also an inline grid: `display: -ms-inline-grid`. Unlike the block level grid, the inline variant allows you to place several grids on the same line. This is shown in the InlineGrid example for this chapter, where we have three `div` elements in the HTML that can be toggled between inline (the default) and block level models:

```
//Within the activated handler
document.getElementById("chkInline").addEventListener("click", function () {
    setGridStyle(document.getElementById("chkInline").checked);
});

setGridStyle(true);

//Elsewhere in default.js
function setGridStyle(inline) {
    var gridClass = inline ? "inline" : "block";

    document.getElementById("grid1").className = gridClass;
    document.getElementById("grid2").className = gridClass;
    document.getElementById("grid3").className = gridClass;
}

/* default.css */
.inline {
```

```

    display: -ms-inline-grid;
}

.block {
    display: -ms-grid;
}

```

When using the inline grid, the elements appear as follows:

Cell 1-1	Cell 1-2	
Cell 2-1	Cell 2-2	

Cell 1-1	Cell 1-2	
Row 2 (spanning columns)		

Cell 1-1	Cell 1-2	Column 3 (spanning rows)
Cell 2-1	Cell 2-2	

When using the block level grid, we see this instead:

Cell 1-1	Cell 1-2
Cell 2-1	Cell 2-2

Cell 1-1	Cell 1-2
Row 2 (spanning columns)	

Cell 1-1	Cell 1-2	Column 3 (spanning rows)
Cell 2-1	Cell 2-2	

Fonts and Text Overflow

As discussed earlier, typography is an important design element for WinRT apps, and for the most part the standard font styles using Segoe UI are already defined in the default WinJS stylesheets. In the Windows SDK there is a very helpful [CSS typography sample](#) that compares the HTML header elements and the `win-type-*` styles, font fallbacks, and using bidirectional fonts (left to right and right to left directions).

Speaking of fonts, custom font resources using the `@font-face` rule in CSS are allowed in WinRT apps. For local context pages, the `src` property for the rule must refer to an in-package font file (that is, a URI that begins with `/` or `ms-appx:///`). Pages running in the web context can load fonts from remote sources.

Another piece of text and typography is dealing with text that overflows its assigned region. You can use the CSS `text-overflow: ellipsis;` style to crop the text with a `...`, and the WinJS stylesheets

contain the `win-type-ellipsis` class for this purpose. In addition to setting `text-overflow`, this class also adds `overflow: hidden` (to suppress scrollbars) and `white-space: nowrap`. It's basically a style you can add to any text element when you want the ellipsis behavior.

The W3C specifications on text overflow, <http://dev.w3.org/csswg/css3-ui/#text-overflow>, is a helpful reference as to what can and cannot be done here. One of the limitations of the current spec is that multiline wrapping text doesn't work with ellipsis. That is, you can word-wrap with the `word-wrap: break-word` style, but it won't cooperate with `text-overflow: ellipsis` (word-wrap wins). I also investigated whether flowing text from a multiline CSS region (see next section) into a single-line region with ellipsis would work, but `text-overflow` doesn't apply to regions. So at present you'll need to shorten the text and insert ellipsis manually if it spans multiple lines.

For a small demonstration of ellipsis and word-wrapping, see the `CenteredText` example for this chapter.

Multicolumn Elements and Regions

Translating the multicolumn flow of content that we're so accustomed to in print media has long been a difficult proposition for web developers. While it's been easy enough to create elements for each column, there was no inherent relationship between the content in those columns. As a result, developers have had to programmatically determine what content could be placed in each element, accounting for variations like font size or changing the number of columns based on the screen width or changes in device orientation.

CSS3 provides for doing multicolumn layout within an element (see <http://www.w3.org/TR/css3-multicol>). With this, you can instruct a single element to lay out its contents in multiple columns, with specific control over many aspects of that layout. The specific styles supported for WinRT apps (with no pesky little vendor prefixes!) are as follows:

CSS Styles	JavaScript Property (<code>element.style.</code>)
<code>column-width</code> and <code>column-count</code> (<code>columns</code> is the shorthand)	<code>columnWidth</code> , <code>columnCount</code> , and <code>columns</code>
<code>column-gap</code> , <code>column-fill</code> , and <code>column-span</code>	<code>columnGap</code> , <code>columnFill</code> , and <code>columnSpan</code>
<code>column-rule-color</code> , <code>column-rule-style</code> , and <code>column-rule-width</code> (<code>column-rule</code> is the shorthand for separators between columns)	<code>columnRuleColor</code> , <code>columnRuleStyle</code> , and <code>columnRuleWidth</code> (<code>columnRule</code> is the shorthand)
<code>break-before</code> , <code>break-inside</code> , and <code>break-after</code>	<code>breakBefore</code> , <code>breakInside</code> , and <code>breakAfter</code>
<code>overflow: scroll</code> (to display scrollbars in the container)	<code>Overflow</code>

The reference documentation for these can be found on [Multi-column layout](#).

Again, Blend provides a great environment to explore how these different styles work. If you're placing a multicolumn element within a variable-size grid cell, you can set `column-width` and let the layout engine add and remove columns as needed, or you can use media queries or JavaScript to set `column-count` directly.

CSS3 multicolumn again only applies to the contents of a single element. While highly useful, it does impose the limitation of a rectangular element and rectangular columns (spans aside). Certain apps like magazines need something more flexible, such as the ability to flow content across multiple elements with more arbitrary shapes, and columns that are offset from one another. These relationships are illustrated in Figure 6-8, where the box in the upper left might be a title, the inset box might contain an image, and the text content flows across two irregular columns.

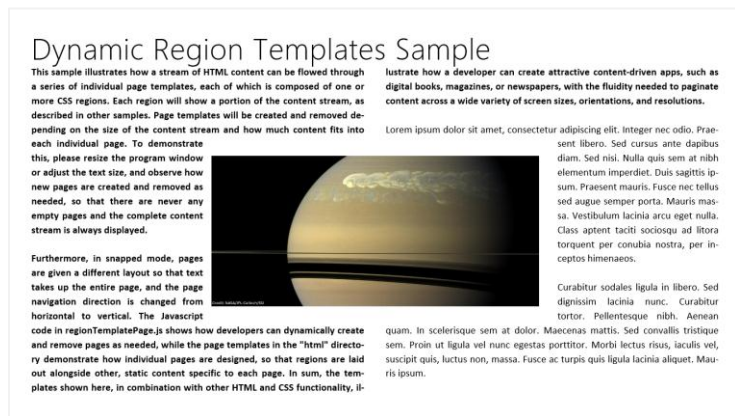


Figure 6-8 Using CSS regions to achieve a more complex layout with irregular text columns.

To support irregular columns, CSS Regions (see <http://dev.w3.org/csswg/css3-regions/>) are coming online and are supported in WinRT apps (see [Regions](#) reference). Regions allow arbitrarily (that is, absolutely) positioned elements to interact with inline content. In Figure 6-8, the image would be positioned absolutely on the page and the column content would flow around it.

The key style for a positioned element is the `float: -ms-positioned` style which should accompany `position: absolute`. Basically that's all you need to do: drop in the positioned element, and the layout engine does the rest. It should be noted that CSS Hyphenation, yet another module, relates closely to all this because doing dynamic layout on text immediately brings up such matters. Fortunately, WinRT apps support the `-ms-hyphens` and the `-ms-hyphenation-*` styles (and their equivalent JavaScript properties). The hyphenation spec is located at <http://www.w3.org/TR/css3-text/>; documentation for WinRT apps is found on the [Text styles reference](#).

The second part of the story consists of named flows and region chains (which are also part of the Regions spec). These provide the ability for content to flow across multiple container elements, as shown in Figure 6-9. Region chains can also allow the content to take on the styling of a particular container, rather than being defined at the source. Each container, in other words, gets to set its own

styling and the content adapts to it, but commonly all the containers share similar styling for consistency.

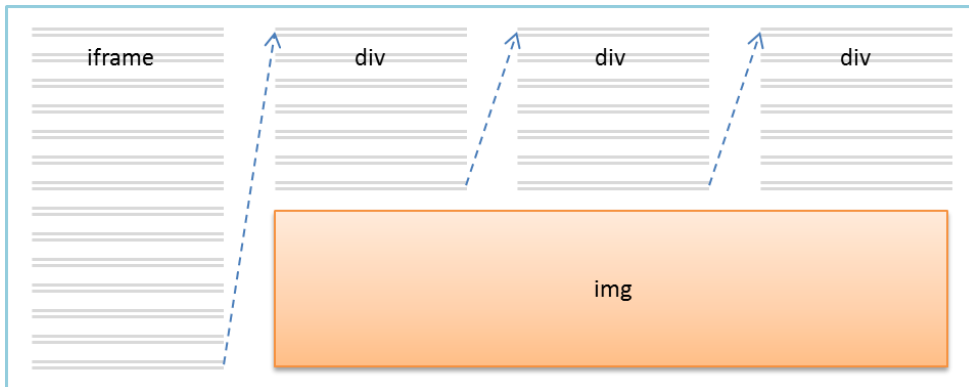


Figure 6-9 CSS region chains to flow content across multiple elements.

How this all works is that the source content is defined by an `iframe` that points to an HTML file (and the `iframe` can be in the web or local context, of course). It's then styled with `-ms-flow-into: <element>` (`msFlowInfo` in JavaScript) where `<element>` is the id of the first container:

```
<!-- HTML -->
<iframe id="s1-content-source" src="/html/content.html"></iframe>
<div class="s1-container"></div>
<div class="s1-container"></div>
<div class="s1-container"></div>

/* CSS */
#s1-content-source {
    -ms-flow-into: content;
}
```

Note that `-ms-flow-into` prevents the `iframe` content from displaying on its own.

Container elements can be any *nonreplaced* element—that is, any element whose appearance and dimensions are not defined by an external resource, such as `img`—and can contain content between its opening and closing tags, like a `div` (the most common) or `p`. Each container is styled with `-ms-flow-from: <element>` (`msFlowFrom` in JavaScript) where the `<element>` is the first container in the flow. The layout then happens in the order elements appear in the HTML (as above):

```
.s1-container {
    -ms-flow-from: content;
    /* Other styles */
}
```

This simple example was taken from the [Static CSS Regions sample](#) in the Windows SDK, which also provides a few other scenarios. There are two other applicable projects here as well, the [Dynamic CSS Regions sample](#) and the [Dynamic CSS Region templates sample](#), where the latter is the source for

Figure 6-8 above. In all these cases, be aware that styling for regions is limited to properties that affect the container and not the content—content styles are drawn from the `iframe` HTML source. This is why using `text-overflow: ellipsis` doesn't work, nor will `font-color` and so forth. But styles like `height` and `width`, along with borders, margin, padding, and other properties that don't affect the content can be applied.

What We've Just Learned

- Layout that is consistent with Windows 8 design principles—specifically the silhouette and typography—helps users focus immediately on content rather than having to figure out each specific app.
- The principle of “content before chrome” allows content to use 75% or more of the display space rather than 25% as is common with chrome-heavy desktop or web applications.
- In some cases, such as a home or hub page of an app with varied and content that does not come from a single collection, it's best to just use plain HTML/CSS layout rather than using a control.
- Pannable HTML sections can use snap points to automatically stop panning at particular intervals within the content.
- The CSS grid is clearly the most useful mechanism for adaptive page-level layout, and it can also be used inline. The CSS flexbox is most useful for inline content, though it has uses at the page level as well, as for centering content vertically and horizontally.
- Every page of an app (including the extended splash screen) can encounter all four view states, so an app design must show how those states are handled. Media queries and the Media Query Listener API can be used to handle the view states declaratively and programmatically.
- Apps can specify a preferred orientation in their manifest and also lock the orientation at run time.
- The `window.onresize` event is best for knowing when the window size has changed, due to view states and/or changes in screen size and pixel density.
- Handling varying screen sizes is accomplished either through a grid-based adaptive layout or a fixed layout utilizing the `WinJS.UI.ViewBox` control that does automatic scaling of its content.
- The chief concern with pixel density is providing graphics that scale well. This means either using vector graphics or providing scaled variants of each raster graphic.

- WinRT apps can take advantage of a wide range of CSS 3 options, including the grid, flexbox, transforms, multicolumn text, and regions.

Chapter 7

Commanding UI

For consumers coming anew to Windows 8 and WinRT apps, one of their first reactions will likely be “Where are the menus? Where is the ribbon? How do I tell this app to do something with the items I selected from a list?” This will be a natural response until users become more accustomed to where commands live, giving another meaning, albeit a mundane one, to the dictum “Blessed are those who have not seen, and yet believe!”

With the design principle of “content before chrome,” UI elements that exist solely to invoke actions and don’t otherwise contain meaningful content fall into the category of “chrome.” As such, they are generally kept out of sight until needed, as are system-level commands like the Charms bar. The user indicates his or her desire for those commands through an appropriate gesture. A swipe on the top or bottom edge of the display, a right mouse button click, or the Win+Z key combination brings up app-specific commands at the top and bottom. A swipe on the left edge of the display, a mouse click on the upper left corner, or Win+Tab allows for switching between apps. And a swipe on the right edge of the display, a mouse click on the upper-right or lower-right corner, or Win+C reveals the Charms bar. (Win+Q, Win+H, and Win+i open the Search, Share, and Settings charms directly.) In the latter case, an app responds to the different charms through particular contracts, as we’ll see in a number of the chapters that follow.

App-specific commands, for their part, are generally provided through an app bar control: [WinJS.UI.AppBar](#). In many ways, the app bar is the equivalent of a menu and ribbon for WinRT apps, because you can create all sorts of UI within it and even show menu elements. Menus, supplied by the [WinJS.UI.Menu](#) control, can also pop up from specific points on the app’s main display, such as a menu attached to a header.

The app bar and menus are specific instances of the more generic [WinJS.UI.Flyout](#) control, which can be used directly for messages or actions that the user can cancel or ignore; such flyouts are dismissed simply by clicking or tapping outside the flyout’s window. (This is like pressing a Cancel button.) For important messages that require action—that is, where the user must choose between a set of options—apps employ [WinJS.UI.MessageDialog](#). Dialog boxes are a familiar concept from the world of desktop applications, of course, and have long been used for collecting all kinds of information and adjusting app settings. In WinRT app design, however, dialog boxes are used only to ask a question and get a simple answer, or just to inform the user of some condition. Settings are specifically handled through the Settings charm, as we’ll see in Chapter 8, “State, Settings, Files, and Documents.”

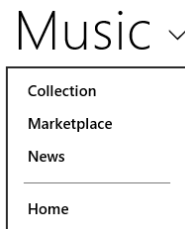
An important point with all of these command controls is that they don’t participate in page layout: they instead “fly out” and remain on top of the current page. This means we thankfully don’t need to worry about their impact on layout...with one small exception that I’ll keep secret for now.

To begin with, though, let's take a step back to think about an app's commands as a whole, and where those commands are ideally placed.

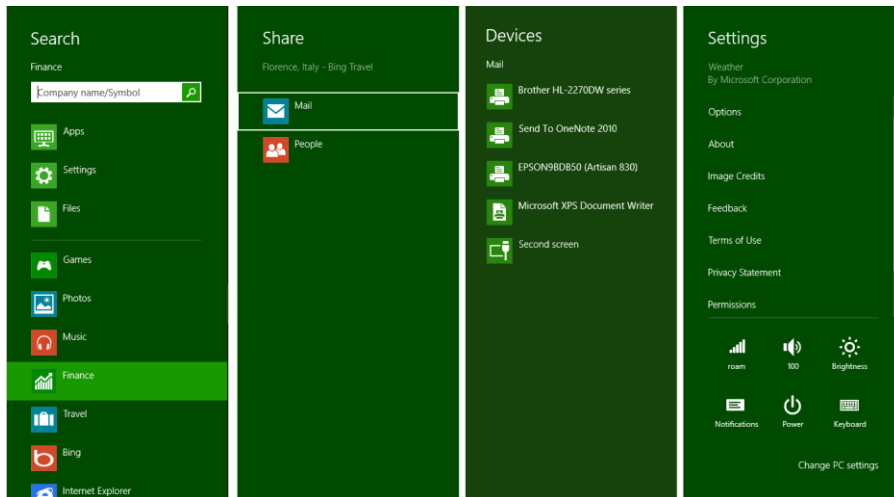
Where to Place Commands

The placement of commands is really quite central to WinRT app design. Unlike the guidelines—or lack thereof!—for desktop application commands, which has resulted in quite a jumble, the Windows Developer Center offers two rather extensive topics on this subject: [Commanding design](#) and [Choosing the right UI surfaces](#). These are must-reads for any designer working on an app, because they describe the different kinds of commanding UI and how to gain the best smiling accolades from Windows 8 design pundits. These are also good topics for developers because they can give you some idea of what you might expect from your designers. Let's review that guidance, then, as an introductory tour to the various options:

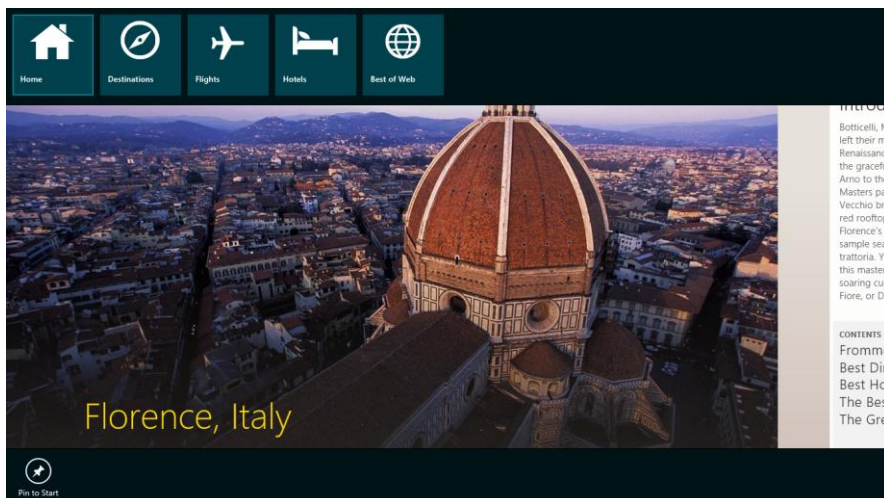
- The user should be able to complete their most important scenarios using just the *app canvas*, so commands that are essential to a workflow should appear directly on-screen. The overall purpose here is to minimize the distraction of unnecessary commands. Nonessential commands should be kept out of view, except for navigation options that can be placed in a drop-down header menu like this:



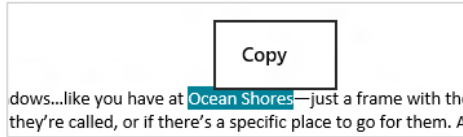
- Always use *Charms* for common app commands where possible. That is, instead of supplying your own search control, use the Search charm (except when the app has a much richer search UI with additional criteria beyond keywords). Instead of supplying individual commands to share with specific targets such as email apps, your contacts, social network apps, and the like, use the Share charm. Instead of supplying your own Print commands, rely on the Device charm. And instead of creating pages within your navigation hierarchy for app settings, help, About, permissions, license agreements, privacy statements, and login/account management, simplify your life and use the Settings charm! (Refer also to “Sidebar: Logins and License Agreements.”) Examples of these are shown in the image below, which also illustrates that many app commands can leverage the Charms bar, which means less clutter in the rest of your commanding UI. Again, we'll cover how to respond to Charms events in later chapters.



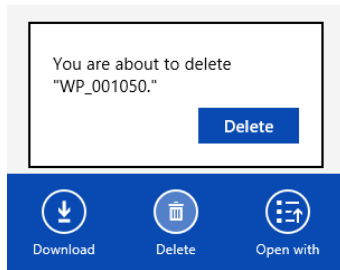
- Commands that can't be placed in Charms and don't need to be on the app canvas are then placed within the *app bar* as shown below in the Travel app; this is the closest analogy to a traditional menu:
- The top app bar is reserved for navigation commands.
- The bottom app bar contains all other commands that are sensitive to the context or selection, as well as global (nonselection) commands. Context and global commands are placed on different sides of the app bar.
- App bar commands can display menus to group related commands and not clutter the app bar itself.



- *Context menus* can provide specific commands for particular content or a selection. For example, selected text typically provides a context menu for clipboard commands, as shown here from the Mail app.



- Confirmations and other questions (including collecting information) that you need to display *in response to a user action* should use a *flyout* control; see [Guidelines and checklist for Flyouts](#). Tapping or clicking outside the control (or pressing ESC) is the same as canceling. Here's an example from the SkyDrive app:



- For blocking events that are not related to a user command but that affect the whole app, use a *message dialog*. A message dialog effectively disables the rest of the app until you pay attention to it! A good example of this is a loss of network connectivity, where the user needs to be informed that some capabilities may not be available until connectivity is restored. User consent prompts for capabilities like geolocation, as shown below from the Maps app, is another place you see message dialogs. Note that a message dialog is used only when the app is in the foreground. Toast notifications, as we'll see in Chapter 13, "Tiles, Notifications, the Lock Screen, and Background Tasks," apply only to background apps.



- Finally, other errors that don't require user action can be displayed either inline (on the app canvas) or through flyouts. See [Laying out your UI: errors](#) for full details; we'll see some examples later on as well.

Where the bottom app bar is concerned, it's also important to organize your commands into sets, as

this streamlines implementation as we'll see in the next section. For full guidance I recommend two additional topics in the documentation: [Guidelines and checklist for app bars](#) and [Commanding Design](#), which provide many specifics on placement, spacing, and grouping. That guidance can be summarized as follows:

- First, make two groups of commands: one with those commands that appear throughout the entire app, regardless of context, and another with those that show only on certain pages. The app bar control is fairly simple to reconfigure at run time for different groups.
- Next, create command sets, such as those that are functionally related, those that toggle view types, and those that apply to selections. Remember again that an app bar command can display a popup menu, as shown below, to provide a list of options and/or additional controls, including longer labels, drop-down lists, checkboxes, radiobuttons, and toggle switches. In this way you can combine closely related commands into a single one that gets more room to play than its little space on the app bar proper.



- For placement, put persistent commands on the right side of the app bar and the most common context-specific commands on the left. After that, begin to populate toward the middle. This recommendation comes from the ergonomic realities of human hands: fingers and thumbs—even on the largest hands of basketball players!—grow only so long and can reach only so far on the screen without having to move one's hand. By placing the most commonly used commands nearest to where a person would be holding a device, as indicated in the image below (from the [Windows 8 Touch Posture topic in the documentation](#)). Those spots are easier to reach (especially by those of us that can't grip a large ball with one hand!) and thus make the whole user experience more comfortable.



- The app bar is always available in all view states, including snapped. It's recommended in snapped view (and sometimes portrait) to limit the commands to 10 so that they can fit into one or two rows.
- Know too that the app bar is not limited to circular command buttons: you can create whatever custom layout you like, which is how top navigation bars are implemented. With any custom layout, make sure that your elements are appropriately sized for touch interaction. More on this—including a small graphic of the aforementioned finger of a basketball player—can again be found on [Guidelines and checklist for app bars](#) as well as [Touch interaction design](#) under “Windows 8 Touch targets.”

Sidebar: Logins and License Agreements

As noted above, Microsoft recommends that login/account management and license agreements/terms-of-use pages are displayed through the Settings charm, where relevant commands are added to the Settings pane that first appears when the charm is invoked. These commands then invoke subsidiary pages with the necessary controls for each of these functions. Of course, sometimes logins and license agreements need some special handling. For example, if your app *requires* a login or license agreement on startup, such controls can be shown on the app's first page or in a flyout. If the user provides a login and/or agrees to the terms of service, the app can continue to run. Otherwise, the app should show a page that indicates that a login or agreement is necessary to do something more interesting than stare at error messages.

If a login is *recommended* but not required, perhaps to enable additional features, you can place those controls directly on the canvas. When the user logs in, you can replace those controls with bits of profile information (user name and picture, for example, as on the Windows Start screen). If, on the other hand, a login is entirely optional, keep it entirely within Settings.

In all cases, commands to view the license agreement, manage one's account or profile, and log in or out should still be available within Settings. Other app bar or on-canvas commands can invoke Settings programmatically, as we'll see in Chapter 8.

The App Bar

After placing the most essential commands on the app canvas, most of your app's commands will be placed in the app bar. Again, the app bar is automatically brought up in response to various user gestures, such as a top or bottom edge swipe, Win+Z, or a right mouse button click. To be specific, whenever you perform one of these gestures, Windows looks for app bar controls on the current page and invokes them—you don't need to process any input events yourself.

For WinRT apps written in HTML and JavaScript, the app bar control is implemented as a WinJS control: [WinJS.UI.AppBar](#). As with all other WinJS controls, you declare an app bar in HTML and instantiate it with a call to [WinJS.UI.process](#) or [WinJS.UI.processAll](#). For a first example, we don't

need to look any farther than some of the Visual Studio/Blend project templates like the Grid App project, where a placeholder app bar is included in default.html (initially commented out):

```
<div id="appbar" data-win-control="WinJS.UI.AppBar">
  <button data-win-control="WinJS.UI.AppBarCommand"
    data-win-options="{id:'cmd', label:'Command', icon:'placeholder'}">
  </button>
</div>
```

The super-exciting result of this markup, using the ui-dark.css stylesheet, is as follows:



Because the app bar is declared in default.html, which is the container for all other page controls, *this same app bar will apply to all the pages in the app*. With this approach you can declare all your commands within a single app bar and assign different classes to the commands that allow you to easily show and hide command sets as appropriate for each page. This also centralizes those commands that appear on multiple pages, such that you can wire up event handlers for them in your app's primary activation code (such as that in default.js).

Alternately, you can declare an app bar within the markup for individual page controls. Since an app bar will still be in the DOM, the Windows gestures will invoke it on each particular page. In the Grid App project, for example, you can move the markup above from default.html into groupedItems.html, groupDetail.html, and itemDetail.html with whatever modifications you like for each page. This might be especially useful if your app's pages don't share many commands in common.

In these cases, each page's `ready` method should take care of wiring up the commands on its particular app bar. Note also that you can add handlers within a page's `ready` method even for a central app bar; it's just a matter of calling `addEventListener` on the appropriate child element within that app bar.

Let's look now at how all this works by using the [HTML AppBar control sample](#). (This chapter's companion content also has a modified version.) We'll start with the basics and the standard command-oriented configuration for app bars, look at how to display menus for some of those commands, and then see how to create custom layouts as is used for a top navigation bar.

Hint Technically speaking, you can declare as many app bars as you want in whatever pages you want, and they'll all be present in the DOM. However, the last one that gets processed in your markup will be the one that's topmost in the z-index (by default) and therefore the one to receive events. Windows does not make any attempt to combine app bars, so because page controls are inserted into the middle of a host page like default.html, an app bar in default.html that's declared after the page control host element will appear on top. At the same time, if the page control's app bar is larger than that in default.html, a portion of it might be visible. The bottom line is, declare app bars *either* in the host page or in a page control, but not both.

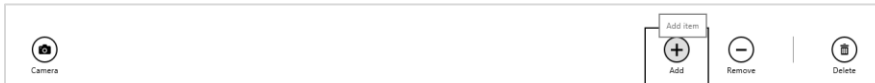
App Bar Basics and Standard Commands

As I just mentioned, an app bar can be declared once for an app in a container page like `default.html` or can be declared separately for each individual page control. The HTML AppBar control sample does the latter, because it provides very distinct app bars for its various scenarios.

Scenario 1 of the sample (`html/create-appbar.html`) declares an app bar with four commands and a separator:

```
<div id="createAppBar" data-win-control="WinJS.UI.AppBar" data-win-options="">
  <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdAdd', label:'Add',
    icon:'add', section:'global', tooltip:'Add item'}">
  </button>
  <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdRemove',
    label:'Remove', icon:'remove', section:'global', tooltip:'Remove item'}">
  </button>
  <hr data-win-control="WinJS.UI.AppBarCommand" data-win-options="{type:'separator',
    section:'global'}" />
  <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdDelete',
    label:'Delete', icon:'delete', section:'global', tooltip:'Delete item'}">
  </button>
  <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdCamera',
    label:'Camera', icon:'camera', section:'selection', tooltip:'Take a picture'}">
  </button>
</div>
```

This appears in the app as follows, using the `ui-light.css` stylesheet, in which we can also see a tooltip, a focus rectangle, and a hover effect on the Add command:



In the markup, the app bar control is declared like any other WinJS control (this is becoming a habit!) using some containing element (a `div`) with `data-win-control="WinJS.UI.AppBar"`. Each page in this sample is loaded with `WinJS.UI.Pages.render` that conveniently calls `WinJS.UI.processAll` to instantiate the app bar. (It is also allowable, as with other controls, to create an app bar programmatically using the `new` operator.)

This example doesn't provide any specific options for the app bar in its `data-win-options`, but there are a number of possibilities:

- `disabled`: if set to `true`, creates an initially disabled app bar; the default is `false`.
- `layout` can be `"commands"` (the default) or `"custom"`, as we'll see in the "Custom App Bars and Navigation Bars" section later.
- `placement` can be either `"top"` or `"bottom"` (the default). We'll use `"top"` for a navigation bar later.
- `sticky` changes the light-dismiss behavior of the app bar. With the default of `false`,

the app bar will be dismissed when you click or tap outside of it. If this is set to `true`, the app bar will stay on the screen until you either change `sticky` to `false` and tap outside or programmatically brush away the control with its `hide` method.

So, if you wanted a sticky navigation bar with a custom layout to appear at the top of the screen, you'd use markup like this:

```
<div id="navBar" data-win-control="WinJS.UI.AppBar"
    data-win-options="{layout:'custom', placement:'top', sticky: true}">
```

Note that having two app bars in a page with different `placement` values will not interfere with each other. Also, the `sticky` property for each placement operates independently. So if you want to implement an appwide top navigation bar, you could declare that within `default.html` (or whatever your top-level page happens to be), and declare bottom app bars in each page control. Again, they're all just elements in the DOM!

As you can see, an app bar control can contain any number of child elements for its commands, each of which *must* be a [WinJS.UI.AppBarCommand](#) control within a `button` or `hr` element or the app bar won't instantiate.

The properties and options of an app bar command are as follows:

- **id** The element identifier, which you can use with `document.getElementById` or the app bar's `getCommandById` method to wire up click handlers.
- **type** One of `"button"` (the default), `"separator"` (which creates a vertical bar), `"flyout"` (which triggers a popup specified with the `flyout` property; see "Command Menu" later on), and `"toggle"` (which creates a button with on/off states). In the latter case, the `selected` property of a command can also be used to set the initial value and to retrieve the state at run time.
- **label** The text shown below for the command button. You always want to use this instead of providing text for the `button` element itself, because such text won't be aligned properly in the control. (Try it and you'll see!) Also, note that this property, along with `tooltip` below, is often localized using `data-win-res` attributes. We'll cover this in Chapter 17, "Apps for Everyone," but for the time being you can look at the `localize-appbar.html` file in the sample (Scenario 8) to see how it works.
- **tooltip** The (typically localized) tooltip text for the command, using the value of `label` as the default. Note that this is just text; using a full HTML-based `WinJS.UI.Tooltip` control here is not supported.
- **icon** Specifies the glyph that's shown in the command. Typically, this is one of the strings from the [WinJS.UI.AppBarIcon enumeration](#), which contains 150 different options from the Segoe UI Symbol font. If you look in the `ui.strings.js` resource file of WinJS you can see how these are defined using codes like `\uE109`—the enumeration, in fact, simply provides friendly names for character codes `\uE100`

through `\uE1E9`. But you're not limited by these. For one thing, you can use any other Unicode escape value `'\uXXXX'` you want from the Segoe UI Symbol font. (Note the single quotes.) You can also use a different font or use your own graphics as described in "Custom Icons" later.³⁸

- `section` Controls the placement of the command. For left-to-right languages (such as English), the default value of `"selection"` places the command on the left side of the app bar and `"global"` places it on the right. For right-to-left languages (such as Hebrew and Arabic), the sides are swapped. These simple choices encourage consistent placement of these two categories of commands (and using any other random value here defaults to `"selection"`). This trains users' eyes to look for the most constant commands on one side and selection-specific commands on the other. Note that the commands in each section are laid out left-to-right (or right-to-left) in the order they appear in your markup.
- `onclick` Can be used to declaratively specify a click handler; remember that any function named here in markup must be marked safe for processing. (See Chapter 4, "Controls and Data Binding" in the "Strict Processing and processAll Functions" section.) Click handlers can also be assigned programmatically with `addEventListener`, in which case the mark is not needed.
- `disabled` Sets the disabled state of a command if `true`; the default is `false`.
- `extraClass` Specifies one or more CSS classes that are attached to the command. These can be used to individually style command controls as well as to create sets that you can easily show and hide, as explained in the "Showing, Hiding, Enabling, and Updating Commands" section later.

If you want to generate commands at run time, you can do so by setting the app bar's `commands` property with an array of JSON `AppBarCommand` descriptors any time the app bar isn't visible (that is, when its `hidden` property is `true`). An array of such descriptors for the Scenario 1 app bar in the sample would be as follows (this is provided in the modified sample included with this chapter; see `js/create_appbar.js`):

```
var appBar = document.getElementById("createAppBar").winControl;

//Set the app bar commands property to populate it
var commands = [
  { id: 'cmdAdd', label: 'Add', icon: 'add', section: 'global', tooltip: 'Add item' },
  { id: 'cmdRemove', label: 'Remove', icon: 'remove', section: 'global', tooltip: 'Remove item' },
  { type: 'separator', section: 'global' },
  { id: 'cmdDelete', label: 'Delete', icon: 'delete', section: 'global', tooltip: 'Delete item' },
```

³⁸ Three notes: First, within `data-win-options` the Unicode escape sequence can also be in the HTML form of `&#xNNNN`; I prefer the JSON form because it has much less ceremony and is less prone to error. Second, you can use the Character Map desktop applet (`charmap.exe`) to examine all the symbols within any particular font. Third, if you need to localize an icon, you can specify the icon property in the `data-win-res` string since the icon property ultimately resolves to a string.

```
{ id: 'cmdCamera', label: 'Camera', icon: 'camera', section: 'selection', tooltip: 'Take a picture' }
];

appbar.commands = commands;
```

When the app bar is created, it will iterate through the `commands` array and create `WinJS.UI.AppBarCommand` controls for each item. If `type` isn't specified or if it's set to `"button"`, `"flyout"`, or `"toggle"`, then the command is a `button` element. A `type` of `"separator"` creates an `hr` element. Note that you should localize the `label`, `tooltip`, and possibly `icon` fields in each command declaration rather than using explicit text as shown here.

You can also use such an array directly within the declarative markup, but this form cannot be localized and is thus discouraged (though I include comments that show how in the modified sample). At the same time, because the value of `commands` in markup is just a string, you can assign its value through data binding with an attribute like this in the app bar element:

```
data-win-bind="{ winControl.commands: myData.commands }"
```

where `myData.commands` can clearly refer to a localized data source. However, this does not work with the `data-win-res` attribute (as we'll see in Chapter 17, "Apps for Everyone" and which is also shown in Scenario 8 of the sample) because the resource string won't be converted to JSON as part of the resource lookup. Attempting to play such a trick would be more trouble than it's worth anyway, so it's best to use either the HTML declarative form or a localized commands array at run time.

Also, be aware that `commands` is a rare example of a *write-only* property: you can set it, but you cannot retrieve the array from an app bar. The app bar uses this array only to configure itself and the array is discarded once all the elements are created in the DOM. At run time, however, you can use the app bar's `getCommandById` method to retrieve a particular command element.

Command Events

Speaking of the command elements, an app bar's `AppBarCommand` controls (other than separators) are all just `button` elements and thus respond to the usual events. Because each command element is assigned the `id` you specify, you can use `getElementById` as usual as a prelude to `addEventListener`. In Scenario 1 of the HTML App Bar control sample, for instance, this code appears in the page's `ready` method:

```
document.getElementById("cmdAdd").addEventListener("click", doClickAdd, false);
document.getElementById("cmdRemove").addEventListener("click", doClickRemove, false);
document.getElementById("cmdDelete").addEventListener("click", doClickDelete, false);
document.getElementById("cmdCamera").addEventListener("click", doClickCamera, false);
```

Although this works, each call to `document.getElementById` traverses the entire DOM and is relatively inefficient. I would recommend that you use the app bar's `getCommandById` method instead, a change I've made throughout the modified sample included with this chapter:

```
//Using the app bar's getCommandById avoids traversing the entire DOM for each button
var appbar = document.getElementById("createAppBar").winControl;
appbar.getCommandById("cmdAdd").addEventListener("click", doClickAdd, false);
```

```
appbar.getCommandById("cmdRemove").addEventListener("click", doClickRemove, false);
appbar.getCommandById("cmdDelete").addEventListener("click", doClickDelete, false);
appbar.getCommandById("cmdCamera").addEventListener("click", doClickCamera, false);
```

Of course, if you specify a handler for each command's `onclick` property in your markup (with each one again having its `supportedForProcessing` property set to `true`), you can avoid all of this entirely!

It should also be obvious that you can wire up events like this from anywhere in your app, and you can certainly listen to any other events you want to, especially when doing custom layouts with other UI. Also, know that the `click` event conveniently handles touch, mouse, and keyboard input alike, so you don't need to do any extra work there. In the case of the keyboard, by the way, the app bar lets you move between commands with the Tab key; Enter or Spacebar will invoke the `click` handler.

App Bar Events and Methods

In addition to the app bar's `getCommandById` method we just saw, the app bar has several other methods and a handful of events. First, the methods:

- `show` displays an app bar if its `disabled` property is `false`; otherwise, the call is ignored.
- `hide` dismisses the app bar.
- `showCommands`, `hideCommands`, and `showOnlyCommands` are used to manage command sets as described in the next section, "Showing, Hiding, Enabling, and Updating Commands."

As for events, there are a total of four that are common to the overlay-style UI controls in WinJS (that is, those that don't participate in layout):

- `[on]beforeshow` occurs before a flyout becomes visible. For an app bar, this is a time when you could set the `commands` property depending on the state of the app at the moment or enable/disable specific commands.
- `[on]aftershow` occurs immediately after a flyout becomes visible. For an app bar, if it's `sticky` property is `true`, you can use this event to adjust the app's layout if you have a scrolling element that might be partially covered otherwise—see below.
- `[on]beforehide` occurs before a flyout is hidden. For an app bar, you'd use this event to hide any supplemental UI created with the app bar and to readjust layout around a `sticky` app bar.
- `[on]afterhide` occurs immediately after a flyout is hidden. For an app bar, this again could be a time to readjust the app's layout if necessary.

You can find an example of using the `show` method along with the `aftershow` and `beforehide` events in Scenario 4 of the HTML AppBar control sample.

The matter with app layout identified above (and what I kept secret in the introduction to this chapter) arises because an app bar overlays and obscures the bottom portion of the page. If that page

contains a scrolling element, an app bar with `sticky` set to `true` will, for mouse users, partly cover a vertical scrollbar and will make a horizontal scrollbar wholly inaccessible. If you're using a sticky app bar with such a page, then—and because Windows Store policy does not look kindly upon discrimination against mouse users!—you should use `aftershow` to reduce the scrolling element's height by the `offsetHeight` or `clientHeight` value of the app bar control, thereby keeping the scrollbars accessible. When the app bar is hidden and `afterhide` fires, you can then readjust the layout. Always use a run-time value like `clientHeight` in these calculations as well, because it accommodates different resolution scales, where the values could be slightly different, and also because the height of an app bar can vary with commands and with view states.

To show this, Scenario 6 of the sample has a horizontally panning `ListView` control that normally occupies most of the page; a scrollbar will appear along the very bottom when the mouse is used. If you select an item, the app bar is made sticky and then shown (see the `doSelectItem` function in `js/appbar-listview.js`):

```
appBar.sticky = true;
appBar.show();
```

The `show` method triggers both `beforeshow` and `aftershow` events. To adjust the layout, the appropriate event to use is `aftershow`, which makes sure the height of the app bar is valid. The sample handles this event in function called `doAppBarShow` (also in `appbar-listview.js`):

```
function doAppBarShow() {
    var listView = document.getElementById("scenarioListView");
    var appBar = document.getElementById("scenarioAppBar");
    var appBarHeight = appBar.offsetHeight;
    // Move the scrollbar into view if appbar is sticky
    if (appBar.winControl.sticky) {
        var listViewTargetHeight = "calc(100% - " + appBarHeight + "px)";
        var transition = {
            property: 'height',
            duration: 367,
            timing: "cubic-bezier(0.1, 0.9, 0.2, 0.1)",
            to: listViewTargetHeight
        };
        WinJS.UI.executeTransition(listView, transition);
    }
}
```

Note The sample on the Windows Developer Center uses `beforeshow` instead of `aftershow`, with the result that sometimes the app bar still has a zero height and the layout is not adjusted properly. To guarantee that the app bar has its proper height for such calculations, use the `aftershow` event as demonstrated in the modified sample included with this chapter's companion content.

Here you can see that the `appBar.offsetHeight` value is simply subtracted from the `ListView`'s `height` with an animated transition. (See Chapter 11, "Purposeful Animations.") The operation is reversed in `doAppBarHide` where the `ListView` height is simply reset to 100% with a similar animation. In this case, the event handler doesn't depend on the app bar's height at all, so it can use either

`beforehide` or `afterhide` events. If, on the other hand, you need to know the size of the app bar for your own layout, use the `beforehide` event.

As an exercise, run Scenario 7 of the SDK sample. Notice how the bottom part of the text region's vertical scrollbar is obscured by the sticky app bar. Try taking some of the code from Scenario 6 to handle `aftershow` and `beforehide` to adjust the text area's height to accommodate the app bar and keep the scrollbar completely visible. And no, I won't be grading you on this quiz: the solution is provided in the modified sample with this chapter.

Showing, Hiding, Enabling, and Updating Commands

In the previous section I mentioned using the `beforeshow` event to configure an app bar's `commands` property such that it contains those commands appropriate to the current page and the page state. This might include setting the `disabled` property for specific commands that are, for example, dependent on selection state. This can be done through the `commands` array, in markup, or again by using the app bar's `getCommandById` method:

```
appbar.getCommandById("cmdAdd").disabled = true;
```

Let me reiterate that the commands that appear on an app bar are specific to each page; it's not necessary to try to maintain a consistent app bar structure across pages. That is, if a command would always be disabled for a particular page, don't bother showing it at all. What's more important is that the app bar for a page is consistent, because it's a really bad idea to have commands appear and disappear depending on the state of the page. That would leave users guessing at how to get the page in the right state for certain commands to appear!

Speaking of changes, it is entirely allowable to modify or update a command at run time, which can eliminate the need to create multiple commands that you alternately show or hide. Since each command on the app bar is just a DOM element, you can really make any changes you want at any time. An example of this is shown in Scenario 3 of the sample where the app bar is initially created with a Play button (custom-icons.html):

```
<button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id:'cmdPlay', label:'Play', icon:'play', tooltip:'Play this song'}">
  </button>
```

This button's click handler uses the `doClickPlay` function in `js/custom-icons.js` to toggle between states:

```
var isPaused = true;

function doClickPlay() {
  var cmd = document.getElementById('cmdPlay');

  if (!isPaused) {
    isPaused = true; // paused
    cmd.winControl.icon = 'play';
    cmd.winControl.label = 'Play';
    cmd.winControl.tooltip = 'Play this song';
```

```

    } else {
        isPaused = false; // playing
        cmd.winControl.icon = 'pause';
        cmd.winControl.label = 'Pause';
        cmd.winControl.tooltip = 'Pause this song';
    }
}

```

Again, the button is just an element in the DOM and updating any of its properties, including styles, will update the element on the screen once you return control to the UI thread.

Now using `beforeshow` for the purpose of adjusting your commands is certainly effective, but you can accomplish the same goal in other ways. The strategy you use depends on the architecture of your app as well as personal preference. From the user's point of view, so long as the appropriate commands are available at the right time, it doesn't really matter how the app gets them there!

Thinking through your approach is especially important when dealing with snapped view, because the recommendation is that you have ten commands or fewer so that the app bar fits on one or two rows. This means that you will want to think through how to adjust the app bar for different view states, perhaps combining multiple commands into a popup menu on a single button.

One approach is to have each page in the app declare and handle its own app bar, which includes pages that create app bars on the fly within their `ready` methods. This makes the relationship between the page content and the app bar very clear and localized. The downside is that common commands—those that appear on more than one page—end up being declared multiple times, making them more difficult to maintain and certainly inviting small inconsistencies like ants to sugar. Nevertheless, if you have very distinct content in your various pages and few common commands, this approach might be the right choice. It is also necessary if your app uses multiple top-level pages rather than one page with page controls, as we discussed in Chapter 3, “App Anatomy and Page Navigation,” because each HTML page has to declare its own app bar anyway.

For apps using page controls, another approach is to declare a single app bar in the top-level page and set its `commands` property within each page control's `ready` method. The drawback here is that because `commands` is a write-only property, you can't declare your common commands in HTML and append your page-specific commands later on, unless you go through the trouble of creating each individual `AppBarCommand` child element within each `ready` method. This kind of code is both tedious to write and to maintain.

Fortunately, there is a third approach that allows you to define a single app bar in your top-level page that contains *all* of your commands, for all of your pages, and then selectively show certain sets of those commands within each page's `ready` method. This is the purpose of the app bar's `showCommands`, `hideCommands`, and `showOnlyCommands` methods.

All three of these methods accept an array of commands, which can be either `AppBarCommand` objects or command id's. `showCommands` makes those commands visible, and can be called multiple times with different sets for a cumulative result. On the opposite side, `hideCommands` hides the specified commands in the app bar, again with cumulative effects. The basic usage of these methods is

demonstrated in Scenario 4 of the sample.

`showOnlyCommands` then combines the two, making specific commands visible while hiding all others. If you declare an app bar with all your commands, you can use `showOnlyCommands` within each page's `ready` method to quickly and easily adjust what's visible. The trick is obtaining the appropriate array to pass to the method. You can, of course, hard-code commands into specific arrays, as Scenario 4 of the sample does for `showCommands` and `hideCommands`. However, if you're thinking that this is A Classic Bad Idea, you're thinking like I'm thinking! Such arrays mean that any changes you make to app bar must happen in both HTML and JavaScript file, meaning that anyone having to maintain your code in the future will surely curse your name!

A better path to happiness and long life is thus to programmatically obtain the necessary arrays from the DOM, using the `extraClass` property on each command to effectively define command sets. This enables you to call `querySelectorAll` to retrieve those commands that belong to a particular set.

Consider the following app bar definition, where for the sake of brevity I've omitted properties like `label`, `icon`, and `section`, as well as any other styling classes:

```
<div id="appbar" data-win-control="WinJS.UI.AppBar" data-win-options="{
  commands:[
    {id:'home', extraClass: 'menuView gameView scoreView'},
    {id:'play', extraClass: 'menuView gameView scoreView'},
    {id:'rules', extraClass: 'menuView gameView scoreView'},
    {id:'scores', extraClass: 'menuView gameView scoreView'},
    {id:'newgame', extraClass: 'gameView gameSnapView'},
    {id:'resetgame', extraClass: 'gameView gameSnapView'},
    {id:'loadgame', extraClass: 'gameView gameSnapView'},
    {id:'savegame', extraClass: 'gameView gameSnapView'},
    {id:'hint', extraClass: 'gameView gameSnapView'},
    {id:'timer', extraClass: 'gameView gameSnapView'},
    {id:'pause', extraClass: 'gameView gameSnapView'},
    {id:'home2', extraClass: 'gameSnapView'},
    {id:'replaygame', extraClass: 'scoreView'},
    {id:'resetscores', extraClass: 'scoreView'}
  ]}">
</div>
```

In the `extraClass` properties we've defined four distinct sets: *menuView*, *gameView*, *gameSnapView*, and *scoreView*. With these in place, a simple call to `querySelectorAll` provides exactly the array we need for `showOnlyCommands`. A generic function like the following can then be used from within each page's `ready` method (or elsewhere) to activate commands for a particular view:

```
function updateAppBar(view) {
  var appbar = document.getElementById("appbar").winControl;
  var commands = appbar.element.querySelectorAll(view);
  appbar.showOnlyCommands(commands);
}
```

With this approach, credit for which belongs to my colleague Jesse McGatha, the app bar is wholly defined in a single location, making it very easy to manage and maintain.

App Bar Styling

The `extraClass` property for commands can, of course, be used for styling purposes as well as managing command sets. It's very simple: whatever classes you specify in `extraClass` are added to the `AppBarCommand` controls created for the app bar.

There are also seven WinJS style classes utilized by the app bar, as described in the following table, where the first two apply to the app bar as a whole and the other five to the individual commands:

CSS class (app bar)	Description
<code>win-appbar</code>	Styles the app bar container; typically this style is used as a root for more specific selectors.
<code>win-commandlayout</code>	Styles the app bar commands layout; apps generally don't modify this style at all.
CSS class (commands)	Description
<code>win-command</code>	Styles the entire <code>AppBarCommand</code> .
<code>win-commandicon</code>	Styles the icon box for the <code>AppBarCommand</code> .
<code>win-commandimage</code>	Styles the image for the <code>AppBarCommand</code> .
<code>win-commandring</code>	Styles the icon ring for the <code>AppBarCommand</code> .
<code>win-label</code>	Styles the label for the <code>AppBarCommand</code> .

Hint To help yourself styling an app bar in Blend, make it `sticky` or add a call to `show` in your page's `ready` method or your app's `activated` event. This makes sure that the app bar is visible and navigable in Blend; it can otherwise be difficult to get the app bar to show within the tool.

Generally speaking, you don't need to override the `win-appbar` or `win-commandlayout` styles directly; instead, you should create selectors for a custom class related to these and then style the particular pieces you need. This can include pseudo-selectors like `button:hover`, `button:active`, and so forth.

Scenario 2 of the HTML Appbar Control sample shows many such selectors in action, in this case to set the background of the app bar and its commands to blue and the foreground color to green (a somewhat hideous combination, but demonstrative nonetheless).

As a basis, Scenario 2 (`html/custom-color.html`) adds a CSS class `customColor` to the app bar:

```
<div id="customColorAppBar" data-win-control="WinJS.UI.AppBar" class="customColor" ...>
```

In `css/custom-color.css` it then styles selectors based on `.win-appbar.customColor`. The following rules, for instance, set the overall background color, the label text color, and the color of the circle around the commands for the `:hover` and `:active` states:

```
.win-appbar.customColor {
    background-color: rgb(20, 20, 90);
}
.win-appbar.customColor .win-label {
    color: rgb(90, 200, 90);
}
.win-appbar.customColor button:hover .win-commandring,
.win-appbar.customColor button:active .win-commandring {
    background-color: rgba(90, 200, 90, 0.13);
    border-color: rgb(90, 200, 90);
}
```

All of this styling, by the way, applies only to the standard command-oriented layout. If you're using a custom layout, the app bar just contains whatever elements you want with whatever style classes you want, so you just handle styling as you would any other HTML.

Custom Icons

Earlier we saw that the `icon` property of an `AppBarCommand` typically comes from the Segoe UI Symbol font. Although this is suitable for most needs, you might want at times to use a character from a different font (some of us just can't get away from Wingdings!) or to provide custom graphics. The app bar supports both.

To use a different font for the whole app bar, simply add a class to the app bar and create a rule based on `win-appbar`:

```
win-appbar.customFont {
    font-family: "Wingdings";
}
```

To change the font of a specific command button, add a class to its `extraClass` property (such as `customButtonFont`) and create a rule with the following selector (as used in Scenario 1 of the modified sample):

```
button.customButtonFont .win-commandimage {
    font-family: "Wingdings";
}
```

To provide graphics of your own, do the following for a 100% resolution scale:

- Create a 160x80 pixel png sprite image with a transparent background, saving the file with the `.scale-100` suffix in the filename.
- This sprite is divided into two rows of four columns—that is, 40x40 pixel cells. The top row is for the light styling theme, and the bottom is for the dark theme.
- Each row has four icons for the following button states, in this order from left to right:

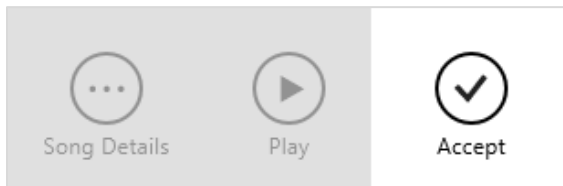
default (rest), hover, pressed (active), and disabled.

- Each image is centered in its respective 40x40 cell, but remember that a ring will be drawn around the icon, so generally keep the image in the 20–30 pixel range vertically and horizontally. It can be wider or taller in the middle areas, of course, where the ring is widest.

For other resolution scales, multiply the sizes by 1.4 (140%) and 1.8 (180%) and use the `.scale-140` and `.scale-180` suffixes in the image filename.

To use the custom icon, point the command's `icon` property to the base image URI (without the `.scale-1x0` suffixes)—for instance, `icon: 'url(images/icon.png)'`.

Scenario 3 of the HTML AppBar Control sample demonstrates custom icon graphics for an Accept button:



The icon comes from a file called `accept.png`, which appears something like this—I've adjusted the brightness and contrast and added a border so that you can see each cell clearly:



The declaration for the app bar button then appears as follows (some properties omitted for brevity):

```
<button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id:'cmdAccept', label:'Accept', icon:'url(images/accept.png)'}">
```

Note that although the sample doesn't have variations of the icon for resolution scales, it does provide variants for high contrast themes, an important accessibility consideration that we'll come back to in Chapter 17. For this reason, the `button` element includes `style="-ms-high-contrast-adjust:none"` to override automatic adjustments for high contrast.

Command Menus

The next aspect of an app bar we need to explore in a little more depth are those commands whose

`type` property is set to `flyout`. In this case the command's `flyout` property must identify a `WinJS.UI.Flyout` object or a `WinJS.UI.Menu` control (which is a flyout). As noted before, flyout/popup menus like this are used when there are too many related commands cluttering up the basic app bar, or when you need other types of controls that aren't quite appropriate on the app bar itself. It's said, though, that if you're tempted to use a button labeled "More", "Advanced", or "Other Stuff" because you can't figure out how to organize the commands otherwise, it's a good sign that the app itself is too complex! Seek ways to simplify the app's purpose so that the app bar doesn't just become a repository for randomness.

We'll be covering flyouts more fully a little later in this chapter, but let's see how to use one in an app bar, as demonstrated in Scenario 6 of the [HTML Flyout Control sample](#) (not the app bar sample, mind you!):



In `html/appbar-flyout.html` of this sample we see the app bar button declared as follows:

```
<button data-win-control="WinJS.UI.AppBarCommand"
  data-win-options="{id: 'respondButton', label: 'Respond', icon: 'edit', type: 'flyout',
  flyout: 'respondFlyout'}">
```

The `respondFlyout` element identified here is defined earlier in `html/appbar-flyout.html`; note that such an element must be declared prior to the app bar to make sure it's instantiated *before* the app bar is created:

```
<div id="respondFlyout" data-win-control="WinJS.UI.Menu">
  <button data-win-control="WinJS.UI.MenuCommand" data-win-options="{id: 'alwaysSaveMenuItem',
    label: 'Always save drafts', type: 'toggle', selected: 'true'}">
  </button>
  <hr data-win-control="WinJS.UI.MenuCommand"
    data-win-options="{id: 'separator', type: 'separator'}" />
  <button data-win-control="WinJS.UI.MenuCommand"
    data-win-options="{id: 'replyMenuItem', label: 'Reply'}">
  </button>
  <button data-win-control="WinJS.UI.MenuCommand"
    data-win-options="{id: 'replyAllMenuItem', label: 'Reply All'}">
  </button>
  <button data-win-control="WinJS.UI.MenuCommand"
    data-win-options="{id: 'forwardMenuItem', label: 'Forward'}">
  </button>
```

</div>

It should come as no surprise by now that the menu is just another WinJS control, `WinJS.UI.Menu`, where its child elements define the menu's contents. As all these elements are, once again, just elements in the DOM; their `click` events are wired up in `js/appbar-flyout.js` with the ever-present `addEventListener`. (Again, the sample uses `document.getElementById` to obtain the elements in order to call `addEventListener`; it would be more efficient to use the app bar's `getCommandById` method instead as in the modified app bar sample.)

Each menu item, as you can see, is a `WinJS.UI.MenuCommand` object, and we'll come back to the details later—for the time being, you can see that those items have an `id`, a `label`, and a `type`, very similar to `WinJS.UI.AppBarCommand` objects.

That's pretty much all there is to it—the one added bit is that when a menu item is selected, you'll want to dismiss the menu and perhaps also the app bar (if it's not sticky). This is shown in the sample within `js/appbar-flyout.js` in a function called `hideFlyoutAndAppBar`:

```
function hideFlyoutAndAppBar() {  
    document.getElementById("respondFlyout").winControl.hide();  
    document.getElementById("appBar").winControl.hide();  
}
```

Custom App Bars and Navigation Bars

All this time we've been looking at the *standard commands layout* of the app bar, which is of course the simplest way to use the control. There will be times, however, when the standard commands layout isn't sufficient. Perhaps you want to place more interesting controls on the app bar, especially custom controls (like a color selector). For this you set the app bar's `layout` property to `'custom'` and place whatever HTML you want within the app bar control, styling it with CSS, and wiring up whatever events you need in JavaScript.

A custom layout is also typically used to implement a top navigation bar—that is, the app bar with `placement` set to `'top'`—because command buttons aren't usually the UI you want. We saw an example earlier in the Weather app, and the navigation bar of Internet Explorer provides another:



Our good friend the HTML AppBar Control sample again delivers an example of custom layout, in Scenario 5. In `html/custom-layout.html` we see the markup for a custom top app bar containing arbitrary elements:

```
<div id="customLayoutAppBar" data-win-control="WinJS.UI.AppBar" aria-label="Navigation Bar"  
    data-win-options="{layout:'custom', placement:'top'}">  
    <header aria-label="Navigation bar" role="banner">
```



```

<button id="cmdBack" class="win-backbutton" aria-label="Back">
</button>
<div class="titleArea">
  <h1 class="win-type-xx-large" tabindex="0">
    Page Title</h1>
  </div>
</header>
</div>

```

Admittedly, the result of this example is a little odd—it creates a navigation bar with a typical page header with a back button where each control might have a focus rectangle. I don’t recommend following this design!



As mentioned in the “Tips and Tricks” section in Chapter 4 (under “Control Styling”), you can suppress the focus rectangle with a `<selector>:focus { outline: none; }` rule in CSS. To remove it from the back button, for example, you can add the style to the following rule in `custom-layout.css`:

```

.win-appbar header .win-backbutton {
  margin-left: 39px;
  margin-top: 59px;
  outline: none;
}

```

Notice again how this rule and the others in `css/custom-layout.css` all use the `win-appbar` class as a base selector but only because it’s styling other generic classes like `header` and `win-backbutton`. If you use specific classes in your app bar or navigation bar, you really don’t need the `win-appbar` selector at all.

To implement a navigation bar like that of Internet Explorer or the Weather app, you can certainly use a `ListView` control along with item templates or custom item rendering functions, where you’d wire up `itemInvoked` events to `WinJS.Navigation.navigate` and so forth. Again, there’s nothing particularly special or complicated here: with a custom layout, the app bar is really just a flyout container for other HTML elements.

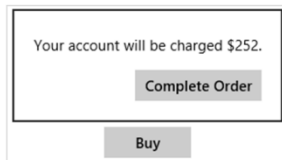
Flyouts and Menus

Going back to our earlier discussion about where to place commands, a flyout control—`WinJS.UI.Flyout`—is used for confirmations, collecting information, and otherwise answering questions in response to a user action. The menu control—`WinJS.UI.Menu`—is then a particular kind of flyout that contains `WinJS.UI.MenuCommand` controls rather than arbitrary HTML. In fact, `WinJS.UI.Menu` is directly derived from `WinJS.UI.Flyout` using `WinJS.Class.define`, so they

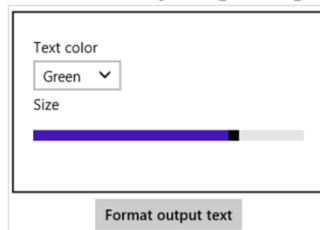
share much in common. As flyouts, they also share some feature in common with the app bar. (Both the app bar and the flyout classes are themselves derived from a [WinJS.UI._Overlay](#) base class that is internal to WinJS.)

Before we look at the details, let's see a number of visual examples from the [HTML Flyout Control sample](#) where we already saw a popup menu on an app bar command. The [WinJS.UI.Flyout](#) controls used in Scenarios 1–4 are shown in Figure 7-1. Notice the variance of content in the flyout itself and how the flyout is always positioned near the control that invoked it, such as the Buy, Login, and Format output text buttons, and the Lorem ipsum hyperlink text. These examples illustrate that a flyout can contain a simple message with a button (Scenario 1, for warnings and confirmations), can contain fields for entering information or changing settings (Scenarios 2 and 3), and can have a title (Scenario 4). Scenario 5, for its part, contains the example of a popup header menu with [WinJS.UI.Menu](#) that we'll see a little later.

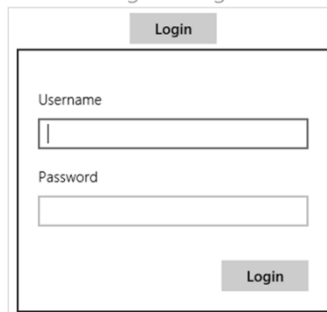
Scenario 1: confirmation flyout



Scenario 3: adjusting settings



Scenario 2: gathering information



Scenario 4: flyout with title

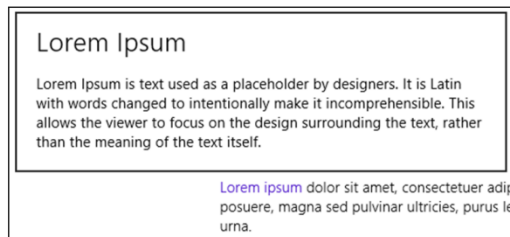


Figure 7-1 Examples of flyout controls from the HTML Flyout control sample.

There are two key characteristics of flyout controls, including menus. One is that flyouts can be dismissed programmatically, like an app bar, when an appropriate control within the flyout is invoked. This is the case with the Complete Order button of Scenario 1 and the Login button of Scenario 2.

The second characteristic, also shared with the app bar, is the light dismiss behavior: clicking or tapping outside the control dismisses it, as does the ESC key, which means light dismiss is the equivalent of pressing a Cancel or Close button in a traditional dialog box. The benefit here is that we don't need a visible button for this purpose, which helps simplify the UI. At the same time, notice in Scenario 3 of Figure 7-1 that there is no OK button or other control to confirm changes you might make in the flyout. With this particular design, changes are immediately applied such that dismissing

the flyout does not reverse or cancel them. If you don't want that kind of behavior, you can place something like an Apply button on the flyout and not make changes until that button is pressed. In this case, dismissing the flyout would cancel the changes.

I'll again encourage you to read the [Guidelines and checklist for Flyouts](#) topic that goes into detail about how and when to use the different designs that are possible with this control. It also outlines when *not* to use the control: for example, to surface errors not related to user action (use a message dialog instead), for primary commands (use the app bar), for text selection context menus, and for UI that is part of a workflow and should be directly on the app canvas. These guidelines also suggest keeping a flyout small and focused (omitting unnecessary controls) and making sure a flyout is positioned close to the object that invoked it. Let's now see how that works in the code.

Note In addition to apps that display a `WinJS.UI.Flyout` on their own, some system APIs (such as that to create a secondary tile) create a system flyout. In these cases, the app will receive a `blur` event, which will cause any light dismiss app bars to be dismissed. To prevent this, set the app bar to `sticky` when using those APIs.

WinJS.UI.Flyout Properties, Methods, and Events

Most of the properties, methods, and events of the `WinJS.UI.Flyout` control are exactly the same as we've already seen for the app bar. The `show` and `hide` methods control its visibility, a `hidden` property indicates its visible state, and same the `beforeshow`, `aftershow`, `beforehide`, and `afterhide` events fire as appropriate. The `afterhide` event is typically used to detect dismissal of the flyout.

Like the app bar, the flyout also has a `placement` property, but it has different values that are only meaningful in the context of the flyout's `alignment` and `anchor` properties. In fact, all three properties are optional parameters to the `show` method because they determine where, exactly, the flyout appears on the screen; the default `placement` and `alignment` can also be set on the control itself because these are optional with `show`. (Note also that if you don't specify an anchor in the `show` method; the `anchor` property must already be set on the control or `show` will throw an exception.)

The `anchor` property identifies the control that invokes the flyout or whatever other operation might bring up a flyout (as for confirmation). The `placement` property then indicates how the flyout should appear in relation to the `anchor`, and the property can contain `'top'`, `'bottom'`, `'left'`, `'right'`, or `'auto'` (the default). Typically, you use a specific `placement` only if you don't want the flyout to possibly obscure important content. Otherwise, you run the risk of the flyout element being shrunk down to fit the available space. The flyout's `content` will remain the same size, mind you, so it means that—ick!—you'll get scrollbars! So, unless you have a really good reason and a note from your doctor, stick with `'auto'` placement so that the control will be placed where it can be shown full size. Along these same lines, remember that in snapped view you have only 320 horizontal pixels to work with, meaning that flyouts you show in that view state should be that size or smaller.

The `alignment` property, for its part, when used with a `placement` of `'top'` or `'bottom'`, determines how the flyout aligns to the edge of the `anchor`: `'left'`, `'right'`, or `'center'` (the default). The

content of the flyout itself is aligned through CSS as with any other HTML.

If you need to style the flyout control itself, you can set styles in the `win-flyout` class, like fonts, default alignments, margins, and so on. As with other WinJS style classes like this, it's best to use `win-flyout` as a basis for more specific selectors unless you really want to style every flyout in the app. Typically, in fact, you also exclude `win-menu` from the rule so that menu flyouts aren't affected by such styling. For example, most of the scenarios in the HTML Flyout control sample, which we'll be looking at next, have rules like this:

```
.win-flyout:not(.win-menu) button,  
.win-flyout:not(.win-menu) input[type="button"] {  
    margin-top: 16px;  
    margin-left: 20px;  
    float: right;  
}
```

Flyout Examples

A flyout control is created like any other WinJS control with `data-win-control` and `data-win-options` attributes and processed by `WinJS.UI.process/processAll`. Flyouts with relatively fixed content will typically be declared in markup where you can use data binding on specific properties of the elements within the flyout. Flyouts that are very dynamic, on the other hand, can be created directly from code by using `new WinJS.UI.Flyout(<element>, <options>)`, and you can certainly change its child elements at any time. It's all just part of the DOM! (Am I repeating myself?)

Like I said before (apparently I am repeating myself), a `WinJS.UI.Flyout` control can contain arbitrary HTML, styled, as always, with CSS. The flyout for Scenario 1 in the sample appears as follows in `html/confirm-action.html` (condensed slightly):

```
<div id="confirmFlyout" data-win-control="WinJS.UI.Flyout" aria-label="{Confirm purchase flyout}">  
    <div>Your account will be charged $252.</div>  
    <button id="confirmButton">Complete Order</button>  
</div>
```

The login flyout in Scenario 2 is similar, and it even employs an HTML form to attach the Login button to the Enter key (`html/collect-information.html`):

```
<div id="loginFlyout" data-win-control="WinJS.UI.Flyout" aria-label="{Login flyout}">  
    <form onsubmit="return false;">  
        <p>  
            <label for="username">Username <br /></label>  
            <span id="usernameError" class="error"></span>  
            <input type="text" id="username" />  
        </p>  
        <p>  
            <label for="password">Password<br /></label>  
            <span id="passwordError" class="error"></span>  
            <input type="password" id="password" />  
        </p>  
        <button id="submitLoginButton">Login</button>  
    </form>  
</div>
```

```

    </form>
</div>

```

The flyout is displayed by calling its `show` method. In Scenario 1, for instance, the button's `click` event is wired to the `showConfirmFlyout` function (js/confirm-action.js), where the Buy button is given as the anchor element. Handling the Complete Order button just happens through a `click` handler attached to that element, and here we want to make sure to call `hide` to programmatically dismiss the flyout. Finally, the `afterhide` event is used to detect dismissal:

```

var bought;

var page = WinJS.UI.Pages.define("/html/confirm-action.html", {
    ready: function (element, options) {
        document.getElementById("buyButton").addEventListener("click", showConfirmFlyout, false);
        document.getElementById("confirmButton").addEventListener("click", confirmOrder, false);
        document.getElementById("confirmFlyout").addEventListener("afterhide", onDismiss, false);
    }

    function showConfirmFlyout() {
        bought = false;
        var buyButton = document.getElementById("buyButton");
        document.getElementById("confirmFlyout").winControl.show(buyButton);
    }

    // When the Buy button is pressed, hide the flyout since the user is done with it.
    function confirmOrder() {
        bought = true;
        document.getElementById("confirmFlyout").winControl.hide();
    }

    // On dismiss of the flyout, determine if it closed because the user pressed the buy button.
    // If not, then the flyout was light dismissed.
    function onDismiss() {
        if (!bought) {
            // (Sample displays a dismissal message on the canvas)
        }
    }
}

```

Handling the login controls in Scenario 2 is pretty much the same, with some added code to make sure that both a username and password have been given. If not, the Login button handler displays an inline error and sets the focus to the appropriate input field:

Username

Password

Password cannot be blank

Login

As the flyout of Scenario 2 is a little larger, the default `placement` of `'auto'` on a 1366x768 display (as in the simulator) makes it appear below the button that invokes it. There isn't quite enough room above that button. So try setting `placement` to `'top'` in the call to `show`:

```
function showLoginFlyout() {
  // ...
  document.getElementById("loginFlyout").winControl.show(loginButton, "top");
}
```

Then you can see how the flyout gets scrollbars because the overall control element is too short:

Username

Password

Login

What was that word I used before? "Ick"?

To move on, Scenario 3 again declares a flyout in markup, where it contains some `label`, `select`, and `input` controls. In JavaScript, though, it listens for change events on the latter and applies those new values to the output element on the app canvas:

```
var page = WinJS.UI.Pages.define("/html/change-settings.html", {
  ready: function (element, options) {
    // ...
    document.getElementById("textColor").addEventListener("change", changeColor, false);
    document.getElementById("textSize").addEventListener("change", changeSize, false);
  }
});
```

```

    }
  });

  // Change the text color
  function changeColor() {
    document.getElementById("outputText").style.color = document.getElementById("textColor").value;
  }

  // Change the text size
  function changeSize() {
    document.getElementById("outputText").style.fontSize =
      document.getElementById("textSize").value + "pt";
  }

```

If this flyout were written to have an Apply button rather than applying the changes immediately, its `click` handler would obtain the current selection and slider values and use them like `changeColor` and `changeSize` do.

Finally, in Scenario 4 we see a flyout with a title, which is just a piece of larger text in the markup; the flyout control itself doesn't have a separate notion of a header:

```

<div id="moreInfoFlyout" data-win-control="WinJS.UI.Flyout" aria-label="{More info flyout}">
  <div class="win-type-x-large">Lorem Ipsum</div>
  <div>
    Lorem Ipsum is text used as a placeholder by designers...
  </div>
</div>

```

The point of this last example is to show that unlike traditional desktop dialog boxes, flyouts don't often need a title because they already have context within the app itself. Dialog boxes in desktop applications need titles because that's what appears in task-switching UI, especially for modal dialogs.

Hint If you find that `beforeshow`, `aftershow`, `beforehide`, or `afterhide` events triggered from a flyout are getting propagated to a containing app bar, which shares the same event names, include a call to `eventArgs.stopPropagation` inside your flyout's handler.

Menus and Menu Commands

What distinguishes a `WinJS.UI.Menu` control from a more generic `WinJS.UI.Flyout` is that a menu expects that all its child elements are `WinJS.UI.MenuCommand` objects, similar to how the standard command layout of the app bar expects `AppBarCommand` objects (and won't instantiate if you declare something else). In fact, the menu control shares other characteristics with the app bar as well as the flyout, such as:

- `show` and `hide` methods.
- `getCommandById`, `showCommands`, `hideCommands`, and `showOnlyCommands`, along with the `commands` property, meaning that you can use the same strategies to manage commands as discussed in "Showing, Hiding, Enabling, and Updating Commands" in

the app bar section, including specifying commands using a JSON array rather than discrete elements.

- `beforeshow`, `aftershow`, `beforehide`, and `afterhide` events.
- `anchor`, `alignment`, and `placement` properties.

The menu also has two styles for its appearance—`win-menu` and `win-command`—that you use to create more specific selectors, as we’ve seen, for the entire menu and for the individual text commands.

`MenuCommand` objects are also very similar to `AppBarCommand` objects. Both share many of the same properties: `id`, `label`, `type` (`'button'`, `'toggle'`, `'flyout'`, and `'separator'`), `disabled`, `extraClass`, `flyout`, `hidden`, `onclick`, and `selected`. Menu commands do not have icons, sections, and tooltips but you can see from `type` that menu items can be buttons (including just text items), checkable items, separators, and also another flyout. In the latter case, the secondary menu will replace the first rather than show up alongside, and to be honest, I’ve yet to see secondary menus used in a real app. Still, it’s supported in the control!

We’ve already seen how to use a flyout menu from an app bar command, which is covered in Scenario 6 of the HTML Flyout controls sample (see the earlier “Command Menus” section). Another primary use case is to provide what looks like drop-down menu from a header element, covered Scenario 5. Here (see `html/header-menu.html`), the standard design is to place a down chevron symbol (``) at the end of the header:

```
<header aria-label="Header content" role="banner">
  <button class="win-backbutton" aria-label="Back"></button>
  <div class="titlearea win-type-ellipsis">
    <button class="titlecontainer">
      <h1>
        <span class="pagetitle">Music</span>
        <span class="chevron win-type-x-large">&#xe099</span>
      </h1>
    </button>
  </div>
</header>
```

Notice that the whole header is wrapped in a `button`, so its `click` handler can display the menu with `show`:

```
document.querySelector(".titlearea").addEventListener("click", showHeaderMenu, false);
```

```
function showHeaderMenu() {
  var title = document.querySelector("header .titlearea");
  var menu = document.getElementById("headerMenu").winControl;
  menu.anchor = title;
  menu.placement = "bottom";
  menu.alignment = "left";
  menu.show();
}
```

The flyout (defined as `headerMenu` in `html/header-menu.html`) appears when you click anywhere

on the header (not just the chevron, as that's just a character in the header text):



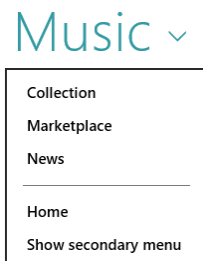
The individual menu commands are just `button` elements themselves, so you can attach `click` handlers to them as you need. As with the app bar, it's best to use the menu control's `getCommandById` to locate these elements as it's much more efficient than `document.getElementById` (as the SDK sample uses...sigh).

To see a secondary menu in action, try adding the following `secondaryMenu` element in `html/header-menu.html` before the `headerMenu` element and adding a `button` within `headerMenu` whose `flyout` property refers to `secondaryMenu`:

```
<div id="secondaryMenu" data-win-control="WinJS.UI.Menu">
  <button data-win-control="WinJS.UI.MenuCommand"
    data-win-options="{id:'command1', label:'Command 1'}"></button>
  <button data-win-control="WinJS.UI.MenuCommand"
    data-win-options="{id:'command2', label:'Command 2'}"></button>
  <button data-win-control="WinJS.UI.MenuCommand"
    data-win-options="{id:'command3', label:'Command 3'}"></button>
</div>

<div id="headerMenu" data-win-control="WinJS.UI.Menu">
  <!-- ... -->
  <button data-win-control="WinJS.UI.MenuCommand"
    data-win-options="{id:'showFlyout', label:'Show secondary menu',
      type:'flyout', flyout:'secondaryMenu'}">
  </button>
</div>
```

Also, go into `css/header-menu.css` and adjust the `width` style of `#headerMenu` to 200px. With these changes, the first menu will appear as follows where the color change in the header is the hover effect:



When you select *Show secondary menu*, the first menu will be dismissed and the secondary one will

appear in its place:

Music ▾

Command 1
Command 2
Command 3

Just to note, another example of a header flyout menu can be found in the Adaptive layout sample we saw in Chapter 6, “Layout.” It’s implemented the same way we see above, with the added detail that it actually changes the page contents in response to a selection.

Context Menus

Besides the flyout menu that we’ve seen so far, there are also context menus as described in [Guidelines and checklist for context menus](#). These are specifically used for commands that are directly relevant to a selection of some kind, like clipboard commands for text, and are invoked with a right mouse click on that selection, a tap, or the context menu key on the keyboard. Text and hyperlink controls already provide these by default. Context menus are also good for providing commands on objects that cannot be selected (like parts of an instant messaging conversation), as app bar commands can’t be contextually sensitive to such items. They’re also recommended for actions that cannot be accomplished with a direct interaction of some kind. However, don’t use them on page backgrounds—that’s what the app bar is for because the app bar will automatically appear with a right-click gesture.

Hint If you process the right mouse button click event for an element, be aware that the default behavior that shows the app bar will be suppressed over that element. Therefore, use the right-click event judiciously, because users will become accustomed to right-clicking around the app to bring up the app bar. Note also that you can programmatically invoke the app bar yourself using its [show](#) method.

The [Context menu sample](#) gives us some context here—I know, it’s a bad pun! In all cases, you need only listen to the HTML `contextmenu` event on the appropriate element; you don’t need to worry about mouse, touch, and keyboard separately. Scenario 1 of the sample, for instance, has a nonselectable *attachment* element on which it listens for the event (`html/scenario1.html`):

```
document.getElementById("attachment").addEventListener("contextmenu", attachmentHandler, false);
```

In the event handler, you then create a [Windows.UI.Popups.PopupMenu](#) object (which comes from WinRT, not WinJS!), populate it with [Windows.UI.Popups.UICommand](#) or [UICommandSeparator](#) objects (that contain an item label and click handler), and then call the menu’s `showAsync` method (`js/scenario1.js`):

```
function attachmentHandler(e) {  
    var menu = new Windows.UI.Popups.PopupMenu();
```

```

menu.commands.append(new Windows.UI.Popups.UICommand("Open with", onOpenWith));
menu.commands.append(new Windows.UI.Popups.UICommand("Save attachment", onSaveAttachment));

menu.showAsync({ x: e.clientX, y: e.clientY }).then(function (invokedCommand) {
    if (invokedCommand === null) {
        // The command is null if no command was invoked.
    }
});
}

```

Notice that the results of the `showAsync` method (and the sample should be calling `done` and not `then` here³⁹) is the `UICommand` object that was invoked; you can examine its `id` property to take further action. Also, the parameter you give to `showAsync` is a `Windows.Foundation.Point` object that indicates where the menu should appear relative to the mouse pointer or the touch point. The menu is placed above and centered on this point.

The `PopupMenu` object also supports a method called `showForSelectionAsync`, whose first argument is a `Windows.Foundation.Rect` that describes the applicable selection. Again, the menu is placed above and centered on this rectangle. This is demonstrated in Scenario 2 of the sample in `js/scenario2.js`:

```

//In the contextmenu handler
menu.showForSelectionAsync(getSelectionRect()).then(function (invokedCommand) { //... }
//...

function getSelectionRect() {
    var selectionRect = document.selection.createRange().getBoundingClientRect();

    var rect = {
        x: getClientCoordinates(selectionRect.left),
        y: getClientCoordinates(selectionRect.top),
        width: getClientCoordinates(selectionRect.width),
        height: getClientCoordinates(selectionRect.height)
    };
    return rect;
}

```

This scenario also demonstrates that you can use a `contextmenu` event handler on text to override the default commands that such controls otherwise provide.

A final note for context menus: because these are created with WinRT APIs and are not WinJS controls, the menus don't exist in the DOM and are not DOM-aware, which explains the use of other WinRT constructs like `Point` and `Rect`. Such is also true of message dialogs, which is our final subject for this chapter.

³⁹ If you're wondering why such consistencies exist, it's because the `done` method didn't originally exist and was introduced mid-way during the production of Windows 8 when it became clear that we needed a better mechanism for surfacing exceptions within chained promises. As a result, numerous SDK samples and code in the documentation still use `then` instead of `done` when handling the last promise in a chain. It still works; it's just that exceptions in the chain will be swallowed, thus hiding possible errors.

Message Dialogs

Our last piece of commanding UI for this chapter is the message dialog. Like the context menu, this flyout element comes not from WinJS but from WinRT via the [Windows.UI.Popups.MessageDialog](#) API. Again, this means that the message dialog simply appears on top of the current page and doesn't participate in the DOM. Message dialogs automatically dim the app's current page and block input events from the app until the user responds to the dialog.

The [Guidelines and checklist for message dialogs](#) topic explains the use cases for this UI:

- To display urgent information that the user must acknowledge to continue, especially conditions that are not related to a user command of some kind.
- Errors that apply to the overall app, as opposed to a workflow where the error is better surfaced inline on the app canvas. Loss of network connectivity is a good example of this.
- Questions that *require* user input and cannot be light dismissed like a flyout. That is, use a message dialog to block progress when user input is essential to continue.

The interface for message dialogs is very straightforward. You create the dialog object with a [new Windows.UI.Popups.MessageDialog](#). The constructor accepts a required string with the message content and an optional second string containing a title. The dialog also has [content](#) and [title](#) properties that you can use independently. In all cases the strings support only plain text.

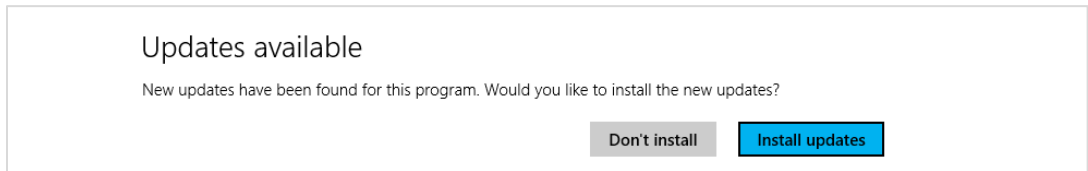
You then configure the dialog through its [commands](#), [options](#), [defaultCommandIndex](#) (the command tied to the Enter key), and [cancelCommandIndex](#) (the command tied to the ESC key).

The [options](#) come from the [Windows.UI.Popups.MessageDialogOptions](#) enumeration where there are only two members: [none](#) (the default, for no special behavior) and [acceptUserInputAfterDelay](#) (which causes the message dialog to ignore user input for a short time to prevent possible clickjacking). This exists primarily for Internet browsers loading arbitrary web content and isn't typically needed for most apps.

The [commands](#) property then contains up to three [Windows.UI.Popups.UICommand](#) objects, the same ones used in context menus. Each command again contains an [id](#), a [label](#), and an [invoked](#) property to which you assign the handler for the command. Note that the [defaultCommandIndex](#) and [cancelCommandIndex](#) properties work on the indices of the [commands](#) array, not the [id](#) properties of those commands. Also, if you don't add any commands of your own, the message dialog will default to a single Close command.

Finally, once the dialog is configured, you display it with a call to its [showAsync](#) method. Like the context menu, the result is the selected [UICommand](#) object that's given to the completed handler you provide to the promise's [done](#) method. Typically, you don't need to obtain that result because the selected command will have triggered its [click](#) event where you normally process those commands.

The [Message dialog sample](#)—one of the simplest samples in the whole Windows SDK!—demonstrates various uses of this API. Scenario 1 displays a message dialog with a title and two command buttons, setting the second command (index 1) as the default. This appears as follows:

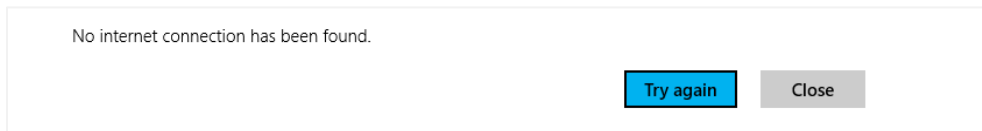


Scenario 2 shows the default Close command with a message and no title:



Scenario 3 is identical to Scenario 1 but uses the completed handler of the `showAsync().done` method to process the selected command.

Finally, Scenario 4 assigns the first command to be the default and marks the second as the cancel command, so the message is dismissed with that command or the ESC key:



And that's really all there is to it!

Improving Error Handling in Here My Am!

To complete this chapter and bring together much of what we've discussed, let's make some changes to Here My Am!, last seen in Chapter 3, to improve its handling of various error conditions.

As it stands right now, Here My Am! doesn't behave very well in a few areas:

- If the Bing Maps control script fails to load from a remote source, the code in `map.html` just throws an exception and the app terminates.
- If we're using the app on a mobile device and have changed our location, there isn't a way to refresh the location on the map other than dragging the pin; that is, the geolocation API is used only on startup.
- When WinRT's geolocation API is trying to obtain a location without a network connection, a several-second timeout period occurs, during which the user doesn't have

any idea what's happening.

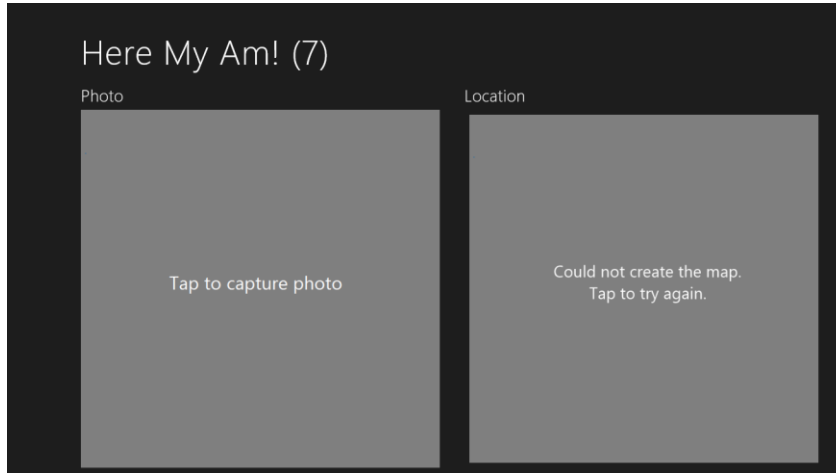
- If our attempt to use WinRT's geolocation API fails, typically due to timeout or network connectivity problems but also possibly due to a denial of user consent, there isn't any way to try again.

The Here My Am! (7) app for this chapter addresses these concerns. First, I've added an error image to the `html/map.html` file (the image is in `html/maperror.png`) so that a failure to load the Bing maps script will display a message in place of the map:

```

```

I've also added a `click` handler to the image that reloads the `iframe` contents with `document.location.reload(true)`. With this in place, I can remove the exceptions that were raised before when the map couldn't be created so that the app doesn't terminate. Here's how it looks if the map can't be created:



To test this, you need to disconnect from the Internet, uninstall the app (to clear any cached map script; otherwise, it will continue to load!), and run the app again. It should hit the error case at the beginning of the `init` method in `map.html`, which shows the error image by removing the default `display: none` style and wiring up the `click` handler. Then reconnect the Internet and click the image, and the map should reload, but if there are continued errors the error message will again appear.

The second problem—adding the ability to refresh our location—is easily done with an app bar. I've added such a control to `default.html` with one command:

```
<div id="appbar" data-win-control="WinJS.UI.AppBar" data-win-options="">
  <button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{id:'cmdRefreshLocation',
    label:'Refresh location', icon:'globe', section:'global', tooltip:'Refresh your location'}">
    </button>
</div>
```

This command is wired up within `pages/home/home.js` in the page control's `ready` method:

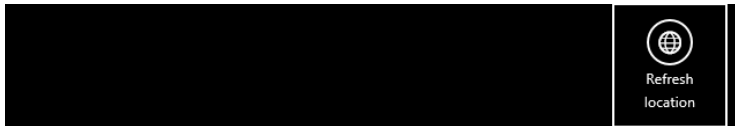
```
var appBar = document.getElementById("appbar").winControl;
appBar.getCommandById("cmdRefreshLocation").addEventListener("click", this.tryRefresh.bind(this));
```

where the `tryRefresh` handler, also in the page control, hides the app bar and calls another new method, `refreshPosition`, where I moved the code that obtains the geolocation and updates the map):

```
tryRefresh: function () {
    //Hide the app bar and retry
    var appBar = document.getElementById("appbar").winControl.hide();
    this.refreshPosition();
},
```

I also needed to tweak the `pinLocation` function within `html/map.html`. Without a location refresh command, this function was only ever called once on app startup. Since it can now be called multiple times, we need to remove any existing pin on the map before adding one for the new location. This is done with a call to `map.entities.pop` prior to the existing call to `map.entities.push` that pins the new location.

The app bar now appears as follows, and we can refresh the location as needed. (If you aren't on a mobile device in your car, try dragging the first pin to another location and then refreshing to see the pin return to your current location.)



For the third problem—letting the user know that geolocation is trying to happen—we can show a small flyout message just before attempting to call the WinRT geolocator's `getGeopositionAsync` call. The flyout is defined in `pages/home/home.html` (our page control) to be centered along the bottom of the map area itself:

```
<div id="retryFlyout" data-win-control="WinJS.UI.Flyout" aria-label="{Trying geolocation}"
    data-win-options="{anchor: 'map', placement: 'bottom', alignment: 'center'}">
    <div class="win-type-large">Attempting to obtain geolocation...</div>
</div>
```

The `refreshPosition` function in `pages/home/home.js` that we just added makes a great place to show this just before calling `getGeopositionAsync`, and we can hide it within the completed and error handlers:

```
refreshPosition: function () {
    document.getElementById("retryFlyout").winControl.show();
    var gl = new Windows.Devices.Geolocation.Geolocator();

    gl.getGeopositionAsync().done(function (position) {
        //...

        //Always hide the flyout
        document.getElementById("retryFlyout").winControl.hide();
    },
```

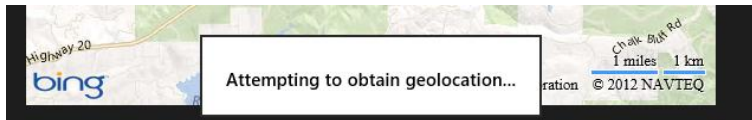
```

    //...
  }, function (error) {
    //...

    //Always hide the flyout
    document.getElementById("retryFlyout").winControl.hide();
  });
},

```

Note that we want to hide the flyout inside the completed and error handlers so that the message stays visible while the async operation is happening. If we placed a single call to hide outside these handlers, the message would flash only very briefly before being dismissed, which isn't what we want. As we've written it, the user will have enough time to see the notice along the bottom of the map (which can be dismissed if you click outside it):



The last piece is to notify the user when obtaining geolocation fails. We could do this with another flyout with a Retry button, or with an inline message as below. We would *not* use a message dialog in this case, however, because the message could appear in response to a user-initiated refresh action. A message dialog might be allowable on startup, but with an inline message combined with the flyout we already added we have all the bases covered.

For an inline message, I've added a floating `div` that's positioned about a third of the way down on top of the map. It's defined in `pages/home/home.html` as follows, as a sibling of the map `iframe`:

```

<div id="locationSection" class="subsection" aria-label="Location section">
  <h2 class="group-title" role="heading">Location</h2>
  <iframe id="map" class="graphic" src="ms-appx-web:///html/map.html" aria-label="Map"></iframe>
  <div id="floatingError" class="win-type-x-large">Unable to obtain geolocation;<br />
    use the app bar to try again.</div>
</div>

```

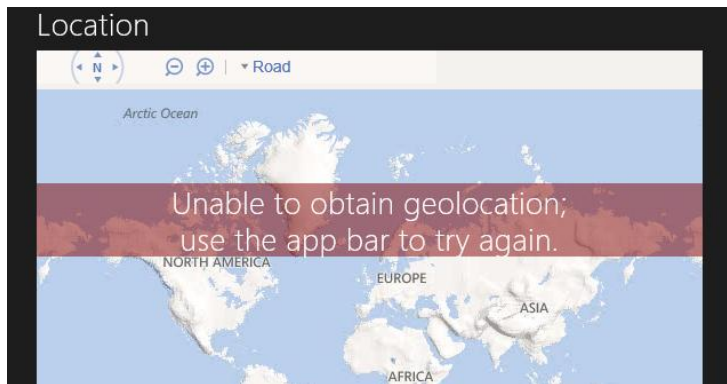
The styles for the `#floatingError` rule in `pages/home/home.css` provide for its placement and appearance:

```

#floatingError {
  display: none;
  float: left;
  -ms-grid-column: 1;
  -ms-grid-row: 2;
  width: 100%;
  text-align: center;
  background-color: rgba(128, 0, 0, 0.5);
  -ms-grid-row-align: start;
  margin-top: 20%;
}

```


Because this is placed in the same grid cell as the map with `float` style, it appears as a nice overlay:



This message will appear if the user denies geolocation consent at startup or allows it but later uses the Settings charm to deny the capability. You can use these variations to test the appearance of the message. It's also possible, if you run the app the first time without network connectivity, for this message to appear on top of the map error image; this is why I've positioned the geolocation error toward the top so that it doesn't obscure the message in the image. But if you've successfully run the app once and you then lose connectivity, the map should still get created because the Bing maps script will have been cached.

With `display: none` in the CSS, the error message is initially hidden, as it should be. If we get to the error handler for `getGeolocationAsync`, we set `style.display` to `block`, which reveals the element:

```
document.getElementById("floatingError").style.display = "block";
```

We again hide the message within the `tryRefresh` function, which is again invoked from the app bar command, so that the message stays hidden unless the error persists:

```
tryRefresh: function () {  
    document.getElementById("floatingError").style.display = "none";  
    //...  
},
```

One more piece we could add is a message dialog if we detect that we lost network connect and thus couldn't update our position. This could be done with the [Windows.Networking.NetworkInformation.onnetworkstatuschanged](#) event, as we'll see in Chapter 14, "Networking." This is a case where a message dialog is appropriate as from the perspective of Here My Am!, such a condition does not arise from direct user action.

Also, it's worth noting that if we used the Bing Maps SDK control in this app, the script we're normally loading from a remote source would exist in our app package, so we'd eliminate the first error case altogether. Because this is a good idea, we'll make this change in the next revision of the app.

What We've Just Learned

- In WinRT app design, commands that are essential to a workflow should appear on the app canvas or on a popup menu from an element like a header. Those that can be placed on the Setting charm should also go there; doing so greatly simplifies the overall app implementation. Those commands that remain typically appear on an app bar or navigation bar, which can contain flyout menus for some commands. Context menus ([Windows.UI.Popups.PopupMenu](#)) can also be used for specific commands on content.
- The [WinJS.UI.Flyout](#) control is used for confirmations and other questions in response to user action; they can also just display a message, collect additional information, or provide controls to change settings for some part of the page. Flyouts are light-dismissed, meaning that clicking outside the control or pressing ESC will dismiss it, which is the equivalent of canceling the question.
- Message dialogs ([Windows.UI.Popups.MessageDialog](#)) are used to ask questions that the user must answer or acknowledge before the app can proceed; a message dialog disables the rest of the app. Message dialogs are best used for errors or conditions that affect the whole app; error messages that are specific to page content should appear inline.
- The app bar is a WinJS control on which you can place standard commands, using the commands layout, or any HTML of your choice, using the custom layout. Custom icons are also possible, using different fonts or custom graphics. An app can have both a top and a bottom app bar, where the top is typically used for navigation and employs a custom layout. App bars can be sticky to keep them visible instead of being light-dismissed.
- The app bar's [showCommands](#), [hideCommands](#), and [showOnlyCommands](#) methods, along with the [extraClass](#) property of commands, make it easy to define an app bar in a single location in the app and to selectively show specific command sets by using [querySelectorAll](#) with a class that represents that set.
- Command menus, as appear from an app bar command or an on-canvas control of some kind, are implemented with the [WinJS.UI.Menu](#) control.
- As an example of using many of these features, the Here My Am! app is updated in this chapter to greatly improve its handling of various error conditions.

Chapter 8

State, Settings, Files, and Documents

It would be very interesting when you travel if every hotel room you stayed in was automatically configured exactly as how you like it—the right pillows and sheets, the right chairs, the right food in the minibar rather than atrociously expensive and obscenely small snack tins. If you're sufficiently wealthy, of course, you can send people ahead of you to arrange such things, but such luxury remains naught but a dream for most of us.

Software isn't bound by such limitations, fortunately. Sending agents on ahead doesn't involve booking airfare for them, providing for their income and healthcare, and contributing to their retirement plans. All it takes is a little connectivity, some cloud services, and voila! All of your settings can automatically travel with you—that is, between the different devices you're using.

This roaming experience, as it's called, is built right into Windows 8 for systemwide settings such as your profile picture, start screen preferences, Internet favorites, your desktop theme, saved credentials, and so forth. When you use a Microsoft account to log into Windows, these settings are securely stored in the cloud and automatically transferred to other Windows 8 devices where you use the same account. I was pleasantly surprised during the development of Windows 8 that I no longer needed to manually transfer all this data when I updated my machine from one release preview to another!

With such an experience in place for system settings, users will expect similar behavior from apps: they will expect that app-specific settings on one device will appropriately roam to the same app installed on other devices. I say "appropriately" because some settings don't make sense to roam, especially those that are particular to the hardware in the device. On the other hand, if I configure email accounts in an app on one machine, I would certainly hope those show up on others! (I can't tell you how many times I've had to set up my four active email accounts in Outlook.) In short, as a user I'll expect that my transition between devices—on the system level and the app level—is both transparent and seamless.

This means, then, that each app must do its part to manage its state, deciding what information is local to a device, what data roams between devices (including roaming documents and other user data through services like SkyDrive), and even what kinds of caching can help improve performance and provide an good offline experience. As I've said with many such functional aspects, the effort you invest in these can make a real difference in how users perceive your app and the ratings and reviews they'll give it in the Windows Store.

Many such settings will be completely internal to an app, but others can and should be directly configurable by the user. In the past, this has given rise to an oft-bewildering array of nested dialog

boxes with multiple tabs, each of which is adorned with buttons, popup menus, and long hierarchies of check boxes and radio buttons. As a result, there's been little consistency in how apps are configured. Even a simple matter of where such options are located has varied between Tools/Options, Edit/Preferences, and File/Info commands, among others!

Fortunately, the designers of Windows 8 recognized that most apps have settings of some kind, so they included Settings on the Charms bar alongside the other near-ubiquitous search, share, and device functions. For one thing, this eliminates the need for users to remember where a particular app's settings are located, and apps don't need to wonder how, exactly, to integrate settings into their overall content flow and navigation hierarchy. That is, by being placed in the Settings charm, settings are effectively removed from an app's content structure, thereby simplifying the app's overall design. The app needs only to provide distinct pages that are displayed when the user invokes the charm, a process that will give us our first taste of an activation path outside the primary launch of an app from a tile.

Clearly, then, an app's state and its Settings UI are intimately connected, as we will see in this chapter. We'll also have the opportunity to look at the storage and file APIs in WinRT, along with some of the WinJS file I/O helpers and other storage options like IndexedDB.

Of course, app data—settings and internal state—is only one part of the story. *User data*—such as documents, pictures, music/audio, and videos—is equally important. For these we'll look at the various capabilities in the manifest that allow an app to work with document and media libraries, as well as removable storage, how to enumerate folder contents with queries, and how the file picker lets the user give consent to other safe areas of the file system (but not system areas and the app data folders of other apps).

Here, too, Windows 8 actually takes us beyond the local file system. The vast majority of data to which a user has access today is not local to their machine but lives online. The problem here has been that such data is typically buried behind the API of a web service, meaning that the user has to manually use a web app to browse data, download and save it to the local file system, and then import it into another app. Seeing this pattern, the Windows 8 designers found another opportunity to introduce a new level of integration and consistency, allowing apps to surface back-end data such that it appears as part of the local file system to other apps. This happens through the file picker contracts, bringing users a seamless experience across local and online data. Here we'll see the consumer side of the story, saving the provider side for Chapter 12, "Contracts."

In short, managing state and providing access to user data, wherever it's located, is one of the most valuable features that apps can provide, and it goes a long way to helping consumers feel that your app is treating them well.

The Story of State

To continue the analogy started in this chapter's introduction, when we travel to new places and stay in

hotels, most of us accept that we'll spend a little time upon arrival unpacking our things and setting up the room to our tastes. On the other hand, we expect the complete opposite from our homes: we expect continuity or *statefulness*. Having moved twice in the last year myself (once to a temporary home while our permanent home was being completed), I can very much appreciate the virtues of statefulness. Imagine that everything in your home got repacked into boxes every time you left, such that you had to spend hours, days, or weeks unpacking it all again! No, home is the place where we expect things to stay put, even if we do leave for a time. I think this exactly why many people enjoy traveling in a motor home!

WinRT apps are intended to be similarly stateful, meaning that they maintain a sense of continuity between sessions, even if the app is suspended and terminated along the way. In this way, apps feel more like a home than a temporary resting place; they become a place where users come to relax with the content they care about. So, the less work they need to do to start enjoying that experience, the better.

We've already discussed the relationship between session state and app lifecycle events in Chapter 3, "App Anatomy and Page Navigation." What we're ready to focus on presently is how to provide the overall sense of statefulness that includes but extends beyond the lifecycle considerations.

Let's first briefly revisit user data again. User data like documents, pictures, music, videos, playlists, and other such data are created and consumed by an app but not dependent on the app itself. User data implies that any number of apps might be able to load and manipulate such data, and such data always remains on a system irrespective of the existence of apps. For this reason, user data itself isn't really part of an app's state. That is, while the *paths* of documents and other files might be remembered as the current file, in the user's favorites, or in a recently used list, the actual *contents* of those files aren't part of that state. User data, then, doesn't have a strong relationship to app lifecycle events either. It's either saved explicitly through a user-invoked command or implicitly on events like *visibilitychanged* rather than *suspending*. Again, the app might remember which file is currently loaded as part of its session state during *suspending*, but the file contents itself should be saved outside of this event since you have only five seconds to complete whatever work is necessary.

In contrast to user data, app data is comprised of everything an app needs to run and maintain its statefulness. App data is also maintained on a per-user basis, is completely tied to the existence of a specific app, and is accessible by that app exclusively. As we've seen earlier in this book, app data is stored in user-specific folders that are wholly removed from the file system when an app is uninstalled. For this reason, never store anything in app data that the user might want outside your app. It also makes sense to avoid using document and media libraries to store state that wouldn't continue to be meaningful to the user if the app is uninstalled.

App data is used to manage the following kinds of state:

- **Session state** The state that an app saves when being suspended to restore it after a possible termination. This includes form data, the page navigation history, and so forth. As we saw in Chapter 3, being restarted after being suspended and terminated is the *only* case in which an app restores session state. Session state is typically saved

incrementally (as the state changes) or within the [suspending](#) event.

- **Local app state** Those settings that are typically loaded when an app is launched. App state includes cached data, saved searches, lists of recently viewed items, and various behavioral settings that appear in the Settings panel like display units, preferred video formats, device-specific configurations, and so on. Local app state is typically saved when it's changed since it's not directly tied to lifecycle events.
- **Roaming app state** App state that is shared between the same app running on multiple Windows 8 devices where the same user is logged in, such as favorites, viewing position within videos, account configurations, URLs for important files on cloud storage locations, perhaps some saved searches or queries, etc. Like local app state, these might be manipulated through the Settings panel. Roaming state is also best saved when values are changed; we'll see more details on how this works later.

An important point to remember with app data is that it's carried forward when a user updates your app. A newer version of an app must be prepared to load and update previous versions of the app data, so you should always include a version number with your app data.

Settings and State

App state may or may not be surfaced directly to the user. Many bits of state are tracked internally within the app or, like a navigation history, might reflect user activity but aren't otherwise presented directly to the user. Other pieces of state, like preferences, accounts, profile pictures, and so forth, can and should be directly exposed to the user, which is the purpose of the Settings charm.

What appears in the Settings charm for an app should be those settings that affect behavior of the app as a whole and are adjusted only occasionally. State that applies only to particular pages or workflows should not appear in Settings: they should be placed directly on the page (the app canvas) or in the app bar, as we've seen in Chapter 7, "Commanding UI." All of these things still comprise app state and are managed as such, but not everything is appropriate for the Settings.

Some examples of good candidates for the Settings charm are as follows:

- Display preferences like units, color themes, alignment grids, and defaults.
- Roaming preferences that allow the user to customize the app's overall roaming experience, such as to keep configurations for personal and work machines separate.
- Account and profile configurations, along with commands to log in, log out, and otherwise manage those accounts and profiles.
- Behavioral settings like online/offline mode, auto-refresh, refresh intervals, preferred video/audio streaming quality, whether to transfer data over metered network connections, the location from which the app should draw data, and so forth.
- A feedback link where you can gather specific information from the user.

- Additional information about the app, such as Help, About, a copyright page, a privacy statement, license agreements, and terms of use. Oftentimes these commands will take the user to a separate website, which is perfectly fine.

I highly recommend that you run the apps that are built into Windows and explore their use of the Settings charm. You're welcome to explore how Settings are used by other apps in the Store as well, but those might not always follow the design guidelines as consistently—and consistency is essential to settings!

Speaking of which, Windows automatically provides commands called Permissions and Rate and Review for all apps. Rate and Review takes the user to the product page in the Windows Store where he or she can, of course, provide a rating and write a review. Permissions, for its part, allows the user to control the app's access to sensitive capabilities like geolocation, the camera, the microphone, and so forth. What appears here is driven by the capabilities declared in the app manifest, and it's where the user can go to revoke or grant consent for those capabilities. Of course, if the app uses no such capabilities, Permissions doesn't appear.

You might have noticed that I've made no mention of showing app updates within Settings. This is specifically discouraged because update notices are provided through the Windows Store directly. This is another way of reducing the kinds of noise with which users have had to contend with in the past, with each app presenting its updates in different ways (and sometimes far too often!).

App Data Locations

Now that we understand what kinds of information make up app state, the next question is, Where is it stored? You might remember from Chapter 1, "The Life Story of an App," that when Windows installs an app for a user (and all WinRT apps are accessible to only the user who installed them), it automatically creates LocalState, TempState, and RoamingState folders within the current user's AppData folder, which are the same ones that get deleted when you uninstall an app. On the file system, if you point Windows Explorer to %localappdata%\packages, you'll see a bunch of folders for the different apps on your system. If you navigate to any of these, you'll see these folders along with one called "Settings," as shown in Figure 8-1 for the built-in Sports app. The figure also shows the varied contents of these folders.

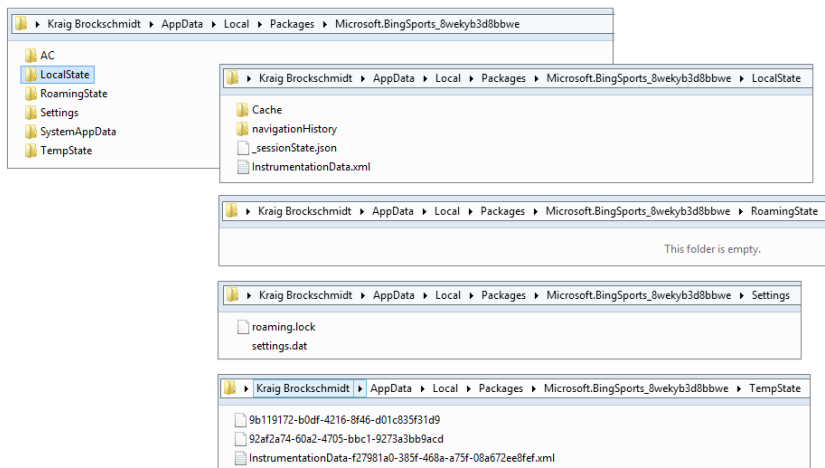


Figure 8-1 The Sports app's AppData folders and their contents.

In the LocalState folder of Figure 8-1 you can see a file named `_sessionState.json`. This is the file where WinJS saves and loads the contents of the `WinJS.Application.sessionState` object as we saw in Chapter 3. Since it's just a text file in JSON format, you can easily open it in Notepad or some other JSON viewer to examine its contents. In fact, if you look open this file for the Sports app, as is shown in the figure, you'll see a value like `{"lastSuspendTime":1340057531501}`. The Sports app (along with News, Weather, etc.) show time-sensitive content, so they save when they were suspended and check elapsed time when they're resumed. If that time exceeds their refresh intervals, they can go get new data from their associated service. In the case of the Sports app, one of its Settings specifically lets the user set the refresh period.

If your app uses any of the HTML5 storage APIs, like local storage, IndexedDB, and app cache, their data will also appear within the LocalState folder.

Note If you look carefully at Figure 8-1, you'll see that all the app data–related folders, including roaming, are in the user's overall AppData/Local folder. There is also a sibling AppData/Roaming folder, but this applies only to roaming user account settings on intranets, such as when a domain-joined user logs in to another machine on a corporate network. This AppData/Roaming folder has no relationship to the AppData/Local.../RoamingState folder for WinRT apps.

Programmatically, you can refer to these locations in several ways. First, you can use the `ms-appdata:///` URI scheme as we saw in Chapter 3, where `ms-appdata:///local`, `ms-appdata:///roaming`, and `ms-appdata:///temp` refer to the individual folders and their contents. (Note the triple slashes, which is a shorthand allowing you to omit the specific package name.) You can also use the object returned from the `Windows.Storage.ApplicationData.current` method, which contains all the APIs you need to work with state.

By the way, you might have some read-only state directly in your app package. With URIs, you can just use relative paths that start with `/`. If you want to open and read file contents directly, you can use the `StorageFolder` object from the `Windows.ApplicationModel.Package.current.installedLocation` property through the same storage APIs that we'll see shortly.

AppData APIs (WinRT and WinJS)

When you ask Windows for the `Windows.Storage.ApplicationData.current` property, what you get is a `Windows.Storage.ApplicationData` object that is completely configured for your particular app. This object contains the following:

- `localFolder`, `temporaryFolder`, and `roamingFolder` Each of these properties is a `Windows.Storage.StorageFolder` object that allows you to create whatever files and additional folder structures you want in these locations (but note the `roamingStorageQuota` below).
- `localSettings` and `roamingSettings` These properties are `Windows.Storage.ApplicationDataContainer` objects that provide for managing a hierarchy of key-value settings pairs or composite groups of such pairs. All these settings are stored in the appdata Settings folder in the settings.dat file.
- `roamingStorageQuota` This property contains the number of kilobytes that Windows will automatically roam for the app (typically 100K); if the total data stored in `roamingFolder` and `roamingSettings` exceeds this amount, roaming will be suspended until the amount falls below the quota.
- `dataChanged` An event indicating the contents of the `roamingFolder` or `roamingSettings` have been synchronized from the cloud; an app should re-read roaming state in this case.
- `signalDataChanged` A method that triggers a `dataChanged` event. This allows you to consolidate local and roaming updates in a single handler for the `dataChanged` event.
- `version` property and `setVersionAsync` method These provide for managing the version stamp on your app data. This version applies to the whole of your app data, local, temp, and roaming together; there are not separate versions for each.
- `clearAsync` A method that clears out the contents of all AppData folders and settings containers. Use this when you want to reinitialize your default state, which can be especially helpful if you've restarted the app because of corrupt state.
- `clearAsync(<locality>)` A variant of `clearAsync` that is limited to one locality (local, temp, and roaming). The locality is identified with a value from `Windows.Storage.ApplicationDataLocality`, such as `Windows.Storage.ApplicationDataLocality.local`. In the case of local and roaming,

the contents of both the folders and settings containers are cleared; temp affects only the TempState folder.

Let's now see how to use the API here to manage the different kinds of app data, which includes a number of WinJS helpers for the same purpose.

Hint The APIs that work with app state will generate events in the Event Viewer if you've enabled the channel as described in Chapter 3 in the "Debug Output, Error Reports, and the Event Viewer" section. Again, make sure that View > Show Analytics and Debug Logs is checked on the menu. Then navigate to Application and Services Log, and expand Microsoft/Windows/AppModel-State, where you'll find Debug and Diagnostic groups.

Settings Containers

For starters, let's look at the `localSettings` and `roamingSettings` properties, which are typically referred to as *settings containers*. You work with these through the `ApplicationDataContainer` API, which is relatively simple. Each container contains four read-only properties: a `name` (a string), a `locality` (again from `Windows.Storage.ApplicationDataLocality`, with `local` and `roaming` being the only values here), and collections called `values` and `containers`.

The top-level settings containers have empty names; the property will be set for child containers that you create with the `createContainer` method (and remove with `deleteContainer`). Those child containers can have other containers as well, allowing you to create a whole settings hierarchy. That said, these settings containers are intended to be used for small amounts of data, typically user settings; any individual setting is limited to 8K and any composite setting (see below) to 64K. With these limits, going beyond about a megabyte of settings implies a somewhat complex hierarchy, which will be difficult to manage and will certainly slow access. So don't be tempted to think of app data settings as a kind of database; other mechanisms like IndexedDB are much better suited for that purpose, and you can write however much data you like as files in the various AppData folders (remembering the roaming limit when you write to `roamingFolder`).

For whatever container you have in hand, its `containers` collection is an `IMapView` object through which you can enumerate its contents. The `values` collection, on the other hand, is just an array (technically an `IPropertySet` object in WinRT, which is projected into JavaScript as an array). So, although the `values` property in any container is itself read-only, meaning that you can't assign some other arbitrary array to it, you can manipulate the contents of the array however you like.

We can see this in the [Application Data sample](#), which is a good reference for many of the core app data operations. Scenario 1, for example (settings.js), shows the simply use of the `localSettings.values` array:

```
var localSettings = Windows.Storage.ApplicationData.current.localSettings;
var settingName = "exampleSetting";
var settingValue = "Hello World";

function settingsWriteSetting() {
```

```

        localSettings.values[settingName] = settingValue;
    }

    function settingsDeleteSetting() {
        localSettings.values.remove(settingName);
    }

```

Many settings, like that shown above, are just simple key-value pairs, but other settings will be objects with multiple properties. This presents a particular challenge: although you can certainly write and read the individual properties of that object within the `values` array, what happens if a failure occurs with one of them? That would cause your state to become corrupt.

To guard against this, the app data APIs provide for *composite settings*, which are groups of individual properties that are guaranteed to be managed as a single unit. (Again, each composite has a 64K limit.) It's like the perfect group consciousness: either we all succeed or we all fail, with nothing in between! That is, if there's an error reading or writing any part of the composite, the whole composite fails; with roaming, either the whole composite roams or none of it roams.

A composite object is created using [Windows.Storage.ApplicationDataCompositeValue](#), as shown in Scenario 2 of the Application Data sample:

```

var roamingSettings = Windows.Storage.ApplicationData.current.roamingSettings;
var settingName = "exampleCompositeSetting";
var settingName1 = "one";
var settingName2 = "hello";

function compositeSettingsWriteCompositeSetting() {
    var composite = new Windows.Storage.ApplicationDataCompositeValue();
    composite[settingName1] = 1; // example value
    composite[settingName2] = "world"; // example value
    roamingSettings.values[settingName] = composite;
}

function compositeSettingsDeleteCompositeSetting() {
    roamingSettings.values.remove(settingName);
}

function compositeSettingsDisplayOutput() {
    var composite = roamingSettings.values[settingName];
}

```

The [ApplicationDataCompositeValue](#) object has, as you can see in the documentation, some additional methods and events to help you manage it.

Composites are, in many ways, like their own kind of settings container, just that they cannot contain additional containers. It's also important to not confuse the two. Child containers within settings are used only to create a hierarchy (refer to Scenario 3 in the sample). Composites, on the other hand, specifically exist to create more complex groups of settings that act like a single unit, a behavior that is not guaranteed for settings containers themselves.

As noted earlier, these settings are all written to the settings.dat file in your app data Settings folder.

It's also good to know that changes you make to settings containers are automatically saved, though there is some built-in batching to prevent excessive disk activity when you change a number of values all in a row. In any case, you really don't need to worry about *when* you save settings; the system will manage those details.

Versioning App State

From Windows' point of view, local, temp, and roaming state are all parts of the same whole and all share the same version. That version number is set with [Windows.Storage.ApplicationData.setVersionAsync](#), the value of which you can retrieve through [Windows.Storage.ApplicationData.version](#) (a read-only property). If you like, you can maintain your own versioning system within particular files or settings. I would recommend, however, that you avoid doing this with roaming data because it's hard to predict how Windows will manage synchronizing slightly different structures. Even with local state, trying to play complex versioning games is, well, rather complex, and probably best avoided altogether.

The version of your app data is also a different matter than the version of your *app*; in fact, there is really no inherent relationship between the two. While the app data version is set with [setVersionAsync](#), the app version is managed through the Packaging section of the app manifest. You can have versions 1.0.0.0 through 4.3.9.3 of the app use version 1.0.0.0 of app data, or maybe version 1.2.1.9 of the app shifts to version 1.0.1.0 of the app data, and version 2.1.1.3 moves to 1.2.0.0 of the app data. It doesn't really matter, so long as you keep it all straight and can migrate old versions of the app data to new versions!

Migration happens as part of the [setVersionAsync](#) call, whose second parameter is a function to handle the conversion. That function receives a [SetVersionRequest](#) object that contains [currentVersion](#) and [desiredVersion](#) properties, thereby instructing your function as to what kind of conversion is actually needed. In response, you should go through all your app data and migrate the individual settings and files accordingly. Once you return from the conversion handler, Windows will assume the migration is complete. Of course, because the process will often involve asynchronous file I/O operations, you can use a deferral mechanism like that we've seen with activation. Call the [SetVersionRequest.getDeferral](#) method to obtain the deferral object (a [SetVersionDeferral](#)), and call its [complete](#) method when all your async operations are done.

An example of version migration can be found in Scenario 7 of the [Application Data sample](#).

Storage Folders and Storage Files

As it is often highly convenient to save app data directly in file, it's high time we start looking more closely at the File I/O APIs for WinRT apps. We have touched on these a little already in the Here My Am! app back in Chapter 3, where we used the local folder's [createFolderAsync](#) and the [StorageFile.copyAsync](#) methods. We're now ready to see the rest of those APIs.

First, however, other APIs like [URL.createObjectURL](#)—working with what are known as *blobs*—make it possible to do many things in an app without having descend to the level of file I/O at

all! We've already seen how to use this to set the `src` of an `img` element, and the same works for other elements like `audio`, `video`, and `iframe`. The file I/O operations involved with such elements is encapsulated within `createObjectURL`. There are other ways to use a blob as well, such as converting a `canvas` element with `canvas.msToBlob` into something you can assign to an `img` element, and obtaining a binary blob from `WinJS.xhr`, saving it to a file, and then sourcing an `img` from that. We'll see some more of this in Chapter 10, "Media," and you can refer to the [Using a blob to save and load content sample](#) in the Windows SDK for more.

For working directly with files, now, let's get a bearing on what we have at our disposal, and you can refer to the [File Access sample](#) for concrete examples.

The core WinRT APIs for files live within the `Windows.Storage` namespace: these are the [StorageFolder](#) and [StorageFile](#) classes, sometimes referred to generically as "storage items" because they are both derived from [IStorageItem](#) and thus share properties like `name`, `path`, `dateCreated`, and `attributes` properties along with the methods `deleteAsync` and `renameAsync`.

With the exception of files obtained from the file picker UI (as we'll see later in this chapter) and the `StorageFile.getFileFromPathAsync` method (see the fifth bullet below), file I/O in WinRT typically starts by obtaining a `StorageFolder` object for the folder in which you want to create, read, or save files and subfolders. That happens through one of these methods or properties:

- [Windows.ApplicationModel.Package.current.installedLocation](#) gets a `StorageFolder` through which you can load data from files in your package (all files therein are read-only).
- `Windows.Storage.ApplicationData.current.localFolder`, `roamingFolder`, or `temporaryFolder` provides `StorageFolder` objects for your app data locations (read-write).
- An app can allow the user to select a folder (or file) directly using the file pickers invoked through [Windows.Storage.Pickers.FolderPicker](#) (plus [FileOpenPicker](#) and [FileSavePicker](#)). This is the preferred way for apps to generally access files when they don't otherwise need to enumerate contents of a library (see next bullet). This is also the only means through which app can access safe (nonsystem) areas of the file system without additional declarations in the manifest.
- [Windows.Storage.KnownFolders](#) provides `StorageFolder` objects for specific libraries (Pictures, Music, Videos, Documents, etc.) in which you can save and load user data according to access declared in your manifest (attempting to obtain a folder without the correct permissions will throw an Access Denied exception). We'll discuss user data and libraries in a later section of this chapter.
- The [Windows.Storage.DownloadsFolder](#) provides a `createFolderAsync` method through which you can obtain a `StorageFolder` in that location. It also provides a

`createFileAsync` method to create a `StorageFile` directly. You would use this API if your app manages downloaded files directly. (Note that `DownloadsFolder` itself provides only these two methods; it is not a `StorageFolder` in its own right.)

- The static method `Windows.Storage.StorageFolder.getFolderFromPathAsync` returns a `StorageFolder` for a given pathname *if and only if* your app already has permissions to access it; otherwise, you'll get an Access Denied exception. A similar static method exists for files called `Windows.Storage.StorageFile.getFileFromPathAsync`, with the same restrictions; `Windows.Storage.StorageFile.getFileFromApplicationUriAsync` opens files with `ms-appx://` (package) and `ms-appdata://` (appdata) URLs. Other schemas are not supported.
- Once a folder or file object is obtained, it can be stored in the `Windows.Storage.AccessCache` mechanism that allows an app to reaccess the same folder or file in the future. This is primarily needed for folders or files selected through the pickers because permission to access the storage item is granted only for the lifetime of that in-memory object. You should always use this API, as demonstrated in Scenario 6 of the File access sample, where you'd otherwise think to save a file path because such paths won't be valid for files provided by another app through the file picker. Even when such a path refers to the local file system, attempting to open that file again using only the path will throw an access denied exception if it exists outside your package, your AppData folders, or those libraries for which you've declared access.

Once you have a `StorageFolder` in hand, you can do the kinds of operations you'd expect: obtain folder properties (including a thumbnail), create and/or open files and folders, and enumerate the folder's contents. With the latter, the API provides for obtaining a list folder contents, of course (see the `getItemsAsync` method), but what you want more often is a partial list of those contents according to certain criteria, along with thumbnails and other indexed file metadata (like music album and track info, picture titles and tags, etc.) that you can use to group and organize the files however you like. This is the purpose of file, folder, and item (file + folder) *queries*, which you manage through the methods `createFileQuery[WithOptions]`, `createFolderQuery[WithOptions]`, and `createItemQuery[WithOptions]`. We already saw a little of this with the FlipView app we built using the Pictures Library in Chapter 5, "Collection and Collection Controls" and we'll return to the subject in the context of user data (the primary scenario for queries) at the end of this chapter.

Tip There are some file extensions that are reserved by the system and won't be enumerated, such as `.lnk`, `.url`, and others; a complete list is found on the [How to handle file activation](#) topic. Also note that the ability to access UNC pathnames requires the Private Networks and Enterprise Authentication capabilities in the manifest along with declarations of the file types you wish to access.

With any given `StorageFolder`, especially for your app data locations, you can clearly create whatever folder structures you like through its `createFolderAsync`/`getFolderAsync` methods, which

gives you more [StorageFolder](#) objects. Within any of those folders you then use the [createFileAsync](#)/[getFileAsync](#) methods to access individual files, each of which you again see as a [StorageFile](#) object.

Each [StorageFile](#) provides relevant properties like [name](#), [path](#), [dateCreated](#), [fileType](#), [contentType](#), [getThumbnailAsync](#), and [attributes](#), of course, along with methods like [copyAsync](#), [deleteAsync](#), [moveAsync](#), [moveAndReplaceAsync](#), and [renameAsync](#) for file management purposes. A file can then be opened in a number of ways depending on the kind of access you need, using these methods:

- [openAsync](#) and [openReadAsync](#) provide random-access byte-reader/writer streams. The streams are [IRandomAccessStream](#) and [IRandomAccessStreamWithContentType](#) objects, respectively, both in the [Windows.Storage.Streams](#) namespace. The first of these works with a pure binary stream; the second works with data+type information, as would be needed with an http response that prepends a content type to a data stream.
- [openSequentialReadAsync](#) provides a read-only [IWindows.Storage.Streams.IInputStream](#) object through which you can read file contents in blocks of bytes but cannot skip back to previous locations. You should always use this method when you simply need to consume the stream as it has better performance than a random access stream (the source can optimize for sequential reads).
- [openTransactedWriteAsync](#) provides a [Windows.Storage.StorageStreamTransaction](#) that's basically a helper object around an [IRandomAccessStream](#) with [commitAsync](#) and [close](#) methods to handle the transactioning. This is necessary when saving complex data to make sure that the whole write operation happens atomically and won't result in corrupted files if interrupted. Scenario 4 of the File Access sample shows this.

The [StorageFile](#) class also provides two static methods, [createStreamedFileAsync](#) and [createStreamedFileFromUriAsync](#), to obtain a [StorageFile](#) that you typically pass to other apps through contracts as we'll see more of in Chapter 12. The utility of these methods is that the underlying file isn't accessed at all until data is first requested from it, *if* such a request ever happens at all.

Pulling all this together now, here's a little piece of code using the raw API we've seen thus far to create and open a "data.tmp" file in our temporary AppData folder, and write a given string to it. This bit of code is in the RawFileWrite example for this chapter. Let me be clear that what's shown here utilizes the lowest-level API in WinRT for this purpose and isn't what you typically use, as we'll see in the next section. It's instructive nonetheless as there are times you need to use something similar:

```
var fileContents = "Congratulations, you're written data to a temp file!";
writeTempFileRaw("data.tmp", fileContents);
```

```
function writeTempFileRaw(filename, contents) {
```

```

var tempFolder = Windows.Storage.ApplicationData.current.temporaryFolder;
var outputStream;

//Egads!
tempFolder.createFileAsync(filename, Windows.Storage.CreationCollisionOption.replaceExisting)
.then(function (file) {
    return file.openAsync(Windows.Storage.FileAccessMode.readWrite);
}).then(function (stream) {
    outputStream = stream.getOutputStreamAt(0);
    var writer = new Windows.Storage.Streams.DataWriter(outputStream);
    writer.writeString(contents);
    return writer.storeAsync();
}).done();
}

```

Good thing we learned about chained async operations a while back! First we create or open the specified file in our app data's temporaryFolder ([createFileAsync](#)), and then we obtain an output stream for that file ([openAsync](#) and [getOutputStreamAt](#)). We then create a [DataWriter](#) around that stream, write our contents into it ([writeString](#)), and make sure it's stored in the file ([storeAsync](#)).

But, you're saying, "You've got to be kidding me! Four chained async operations just to write a simple string to a file! Who designed this API?" Indeed, when we started building the very first WinRT apps within Microsoft, this is all we had, and we asked these questions ourselves! After all, doing some hopefully simple file I/O is typically the first thing you add to your first Hello World app, and this was anything but simple. To make matters worse, at that time we didn't yet have promises for async operations in JavaScript either, so we had to write the whole thing with raw nested operations. Such were the days.

Fortunately, simpler APIs were already available and more came along shortly thereafter. These are the APIs you'll typically use when working with files as we'll see in the next section. It is nevertheless important to understand the structure of the low-level code above because the [Window.Storage.Streams.DataWriter](#) class shown in that code, along with its [DataReader](#) sibling, are very important mechanisms for working with a variety of different I/O streams and are essential for data encoding processes. Having control over the fine details also supports scenarios such as having different components in your app that are all contributing to the file structure. So it's good to take a look at their reference documentation along with the [Reading and Writing data sample](#) just so that you're familiar with their capabilities.

The FileIO, PathIO, and WinJS helper classes (plus FileReader)

Simplicity is a good thing where File I/O is concerned, and the designers of WinRT made sure that the most common scenarios didn't require a long chain of async operations like we saw in the previous section. The [Windows.Storage.FileIO](#) and [PathIO](#) classes provide such a streamlined interface, with the only difference between the two being that the [FileIO](#) methods take a [StorageFile](#) parameter whereas the [PathIO](#) methods take a filename string. Otherwise they offer the same methods called [\[read | write\]BufferAsync](#) (these work with byte arrays), [\[append | read | write\]LinesAsync](#) (these work with arrays of strings), and [\[append | read | write\]TextAsync](#) (these work with singular strings).

In the latter case, the [WinJS.IOHelper](#) class provides an even simpler interface through its [readText](#) and [writeText](#) methods.

Let's see how those work, starting with a few examples from the [File Access sample](#). Scenario 2 shows writing a text string from a control to a file (this code is simplified from the sample for clarity):

```
var userContent = textArea.innerText;

//sampleFile created on startip from Windows.Storage.KnownFolders.documentsLibrary.getFileAsync
Windows.Storage.FileIO.writeTextAsync(sampleFile, userContent).done(function () {
    outputDiv.innerHTML = "The following text was written to '" + sampleFile.name
        + "':<br /><br />" + userContent;
});
```

To compare to the example in the previous section, we can replace all the business with streams and `DataWriters` with one line of code:

```
tempFolder.createFileAsync(filename, Windows.Storage.CreationCollisionOption.replaceExisting)
    .then(function (file) {
        Windows.Storage.FileIO.writeTextAsync(file, contents).done();
    })
```

To make it even simpler, the [WinJS.Application.temp](#) object (an [WinJS.Application.IOHelper](#)) reduces even this down to a single line (which is an `async` call and returns a promise):

```
WinJS.Application.temp.writeText(file, contents);
```

Reading text through the `async` [readText](#) method is equally simple, and WinJS provides the same interface for the [local](#) and [roaming](#) folders along with two other methods, [exists](#) and [remove](#).⁴⁰ That said, these WinJS helpers are available *only* for your AppData folders and not for the file system more broadly; for that you should be using the [FileIO](#) and [PathIO](#) classes.

You also have the HTML5 [FileReader](#) class available for use in WinRT apps, which is part of the [W3C File API specification](#). As its name implies, it's suited only for reading files and cannot write them, but one of its strengths is that it can work both with files and blobs. Some examples of this are found in the [Using a blob to save and load content sample](#).

Encryption and Compression

WinRT provides two additional capabilities that might be very helpful to your state management: encryption and compression.

Encryption is provided through the [Windows.Security.Cryptography](#) and [Windows.Security.Cryptography.Core](#) API. The former contains methods for basic encoding and decoding (base64, hex,

⁴⁰ If you're curious as to why `async` methods like [readText](#) and [writeText](#) don't have `Async` in their names, this was a conscious choice on the part of the WinJS designers to follow existing JavaScript conventions where such a suffix isn't used. The WinRT API, on the other hand, is language-independent and thus has its own convention where the `Async` suffix is included.

and text formats); the latter handles actual encryption according to various algorithms. As demonstrated in the [Secret saver encryption sample](#), you typically encode data in some manner with the `Windows.Security.Cryptography.CryptographicBuffer.convertStringToBinary` method and then create or obtain an algorithm and pass that with the data buffer to `Windows.Security.Cryptography.Core.CryptographicEngine.encrypt`. Methods like `decrypt` and `convertBinaryToString` perform the reverse.

Compression is a little simpler in that its only purpose is to provide a built-in API through which you can make your data smaller (say, to decrease the size of your roaming data). The API for this in [Windows.Storage.Compression](#) is composed of `Compressor` and `Decompressor` classes, both of which are demonstrated in the [Compression sample](#). Understand that this API does *not* work with ZIP files or other standard compression formats—it's merely intended to provide a baseline data compression method. Adjust your expectations accordingly!

Using App Data APIs for State Management

Now that we've seen the nature of the APIs, let's see how they're used for different kinds of app data and any special considerations that come into play.

Session State

As described before, session state is whatever an app saves when being suspended so that it can restore itself to that state if it's terminated by the system and later restarted. Being terminated by the system is again the only time this happens, so what you include in session state should always be scoped to that exact scenario, which is to say, whatever is necessary to give the user the illusion that the app was running the whole time. In some cases, as described in Chapter 3, you might not in fact restore this state, especially if it's been a long time since the app was suspended such that it's unlikely the user would really remember where they left off. That's a decision you need to make for your own app and the experience you want to provide for your customers.

Session state should be saved within the appdata `localFolder` or the `localSettings` object. It should not be saved in a temp area since the user could run the disk cleanup tool while your app is suspended or terminated in which case session state would be deleted (see next section).

The WinJS `sessionState` object internally creates a file called `_sessionState.json` within the `localFolder`, so it follows this same pattern (and the file is just JSON text, so you can examine it any time). You can and should write session state to the `sessionState` object whenever it changes, and really just use `sessionState` as the container for your run-time variables. This way they get saved and reloaded automatically without needing to transfer its contents to other variables.

If you need to store additional values within `sessionState` before its written, do that in your handler for `WinJS.Application.oncheckpoint`. A good example of such data is the navigation stack for your page controls, which is available in `WinJS.Navigation.history`; you could also copy this data to `sessionState` within the `PageControlNavigator.navigated` method (in `navigator.js` as provided by the project templates. In any case, WinJS has its own `checkpoint` handler that is always called last

(after your handler) to make sure that any changes you make to `sessionState` in response to that event are saved.

If you don't use the WinJS `sessionState` object and just use the WinRT appdata APIs directly, you can write your session state whenever you like (including within `checkpoint`), and you'll need to restore it directly within your activated event for `previousExecutionState == terminated`.

It's also a good practice to build some resilience into your handling of session state: if what gets loaded doesn't seem consistent or has some other problem, revert to default session values. Remember too that you can use the `localSettings` container with composite settings to guarantee that groups of values will be stored and retrieved as a unit. You might also find it helpful during development to give yourself a simple command to clear your app state in case things get really fouled up, but then just uninstalling your app will clear all that out as well. At the same time, it's not necessary to provide your users with a command to clear session state: if your app fails to launch after being terminated, the `previousExecutionState` flag will be `notRunning` the next time the user tries, in which case you won't attempt to restore the state.

Similarly, it's not necessary to include a version number in session state. If the user installs an update during the time your app has been suspended and terminated, the `previousExecutionState` value will be reset if the app's version number has changed in any way. If for some reason you don't actually change the version number, for instance, if your update is only a very minor patch, then your session state can carry forward. But in this case it's essentially the same app, so versioning the state isn't an issue.

Sidebar: Using HTML5 `sessionStorage` and `localStorage`

If you prefer, you can use HTML5 `localStorage` object for both session and other local app data; its contents get persisted to the app data `localFolder` as well. The contents of `localStorage` are not loaded until first accessed and are limited to 10MB per app; the WinRT and WinJS APIs, on the other hand, are limited only by the capacity of the file system.

As for the HTML5 `sessionStorage` object, it's not really needed when you're using page controls and maintaining your script context between app pages—your in-memory variables already do the job. However, if you're actually changing page contexts by using `<a>` links and such, `sessionStorage` can still be useful; you can also encode information into URIs as commonly done with web apps.

Both `sessionStorage` and `localStorage` are also useful within `iframe` pages running in the web context, since the WinRT APIs are not available. At the same time, you can load WinJS into a web context page (this is supported), and the `WinJS.Application.local`, `roaming`, and `temp` objects still work. In this case the WinJS objects will be using an in-memory buffer instead of writing to the file system, but all their methods still work.

Local and Temporary State

Unlike session state that is restored only in particular circumstances, local app state is composed of those settings and other values that are *always loaded* when an app is launched. Anything that the user can set directly falls into this category, unless it's also part of the roaming experience, in which case it is still loaded on app startup. Any other cached data, saved searches, recently used items, display units, preferred media formats, and device-specific configurations also fall into this bucket. In short, if it's not pure session state and not part of your app's roaming experience, it's local or temporary state.

All the same APIs we've seen work for this form of state, including all the WinRT APIs, the `WinJS.Application.local` and `temp` objects, and HTML `localStorage`. You can also use the HTML5 IndexedDB APIs (see the next section) and the HTML App Cache feature (see sidebar)—these are just other forms of local app data.

Unlike with session state, it's very important to version-stamp your local and temp app data, because it will always be preserved across an app update (unless temp state has been cleaned up in the meantime). With any app update, then, be prepared to load old versions of your state and make the necessary updates, or simply decide that a version is too old and purge it (`Windows.Storage.ApplicationData.current.clearAsync`) before setting up new defaults.

Generally speaking, local and temp app data are exactly the same—they have the same APIs and are stored in parallel folders. One exception is that temp doesn't support settings and settings containers. The other is that the contents of your temp folder (along with the HTML5 app cache) are subject to the Windows Disk Cleanup tool. This means that your temp data could disappear at any time when the user wants to free up some disk space. For this reason, temp should be used for storage that optimizes your app's performance but not for anything that's critical to its operation. For example, if you have a JSON file in your package that you parse on first startup such that the app starts more quickly afterwards, and you don't make any changes to that data from the app, you might elect to store that in temp. The same is true for graphical resources that you might have fine-tuned for the specific device you're running on; you can always repeat that process from the original resources, so it's another good candidate for temp data. Similarly, if you've acquired data from an online service as an optimization (that is, so that you can just update the local copy incrementally), you can always reacquire it. This is especially helpful for providing an offline experience for your app, though in some cases you might want to let the user choose to save it in local instead of temp (an option that would appear in Settings along with the ability to clear the cache).

Sidebar: HTML5 App Cache

WinRT apps can employ the HTML 5 app cache as part of an offline/caching strategy. It is most useful in `iframe` web context elements where it can be used for any kind of content. For example, an app that reads online books can show such content in an `iframe`, and if those pages include app cache instructions, they'll be saved and available offline. Of course, it will work for any remote page content.

In the local context, the app cache works for nonexecutable resources like images, audio, and video, but not for HTML or JavaScript.

IndexedDB and Other Database Options

Many forms of local app data are well suited to being managed in a database. In WinRT apps, the IndexedDB API is available through the `window.indexedDB` and `worker.indexedDB` objects. For complete details on using this feature, I'll refer you to the [W3C specifications](#), the [Indexed Database API reference](#) for WinRT apps, and the [IndexedDB Sample](#) in the Windows SDK.

Although an IndexedDB database is stored within your app's local app data, be aware that there are some limitations that are imposed because there isn't a means through which the app or the system can shrink a database file and reclaim unused space. Here's a rundown of that and a few other details:

- IndexedDB has a 250MB limit per app and an overall system limit of 12.5% (1/8th) of available storage or 1TB, whichever is less. So it could be true that your app might not have much room to work with anyway, in which case you need to make sure you have a fallback mechanism.
- IndexedDB on Windows 8 has no complex key paths—that is, it does not presently support multiple values for a key or index (multientry).
- By default, access to IndexedDB is given only to HTML pages that are part of the app package and those declared as content URIs. (See the “Local and Web Contexts within the App Host” section at the beginning of Chapter 3.) That is, random web pages you might host in an `iframe` will not be given access, primarily to preserve space within the 250MB limit for those pages you really care about in your app. However, you can grant access to arbitrary pages by including the following tag in your home page and not setting the `iframe src` attribute until the `DOMContentLoaded` or `activated` event has fired:

```
<meta name="ms-enable-external-database-usage" content="true"/>
```

Beyond IndexedDB there are a few other database options for WinRT apps. For a local relational database, try SQLite. This is an API that's suited well for apps written in a language like C#, as described in [Tim Heuer's blog on the subject](#), but fortunately, there is a version called SQL.js, which is [SQLite compiled to JavaScript via Emscripten](#). Very cool! There might also be other JavaScript solutions available in the community.

If the storage limits for IndexedDB are a concern, you might use the [Win32 “Jet” or Extensible Storage Engine \(ESE\) APIs](#) (on which the IndexedDB implementation is built). For this you'll need to write a WinRT Component wrapper in C# or C++ (the general process for which is in Chapter 16, “WinRT Components”), since JavaScript cannot get to those APIs directly.

The same is true for other third-party database APIs. So long as that engine uses only the Win32

APIs allowable for WinRT apps (listed on the [Win32 and COM for WinRT apps](#) page), they'll work just fine.

It's also worth noting that the [OData Library for JavaScript](#) also works great for WinRT apps to access online SQL Servers, because the OData protocol itself just works via REST.

Finally, another option for searchable file-backed data is to use the *system index* by creating a folder named "indexed" in your local AppData folder. The contents of the files in this folder will be indexed by the system indexer and can be queried using Advanced Query Syntax (AQS) with the APIs explained later in "Rich Enumeration with File Queries." You can also do property-based searched for [Windows properties](#), making this approach a simple alternative to database solutions.

Roaming State

The automatic roaming of app state between a user's devices is again one of the most interesting and enabling features of Windows 8. There are few areas where a small piece of technology like this has so greatly reduced the burden on app developers!

It works very simply. First, your app data `roamingFolder` and your `roamingSettings` container behave exactly like their local counterparts. So long as their combined size is less than `Windows.Storage.ApplicationData.current.roamingStorageQuota`, Windows will copy that data to other devices where the same user is logged in has the same app installed; in fact, when an app is installed, Windows attempts to copy roaming data so that it's there when the app is first launched.

If the app is running simultaneously on multiple devices, the last writer of any particular file or setting always wins, and when data has been roaming those apps will receive the `Windows.Storage.ApplicationData.ondatachanged` event. So your app will always read the appropriate roaming state on startup and refresh that state as needed within `datachanged`. You should always employ this strategy too in case Windows cannot bring down roaming state for a newly installed app right away (such as when the user installed the app and lost connectivity). As soon as the roaming state appears, you'll receive the `datachanged` event. Scenario 4 of the [Application Data sample](#) provides a basic demonstration of this.

Deciding what your roaming experience really looks like is really a design question more than a development question. It's a matter of taking all app settings that are not specific to the device hardware (such as settings that are related to screen size, video capabilities, or the presence of particular peripherals or sensors), and thinking through whether it makes sense for each setting to be roamed. A user's favorites, for example, are appropriate to roam *if* they refer to data that isn't local to the device. That is, favorite URIs or locations on a cloud storage service like SkyDrive, FaceBook, or Flickr are appropriate to roam; favorites and recently used files in a user's local libraries are not. The viewing position within a cloud-based video, like a streaming movie, would be appropriate to roam, as would be the last reading position in a magazine or book. But again, if that content is local, then maybe not. Account configurations like email settings are often good candidates, so the user doesn't have to configure the app again on other devices.

At the same time, you might not be able to predict whether the user will really want to roam certain settings. In this case, the right choice is to give the user a choice! That is, include options in your Settings UI to allow the user to customize the roaming experience to their liking, especially as a user might have devices for both home and work use where they do want the same app to behave differently. For instance, with a RSS Reader the user might not want notifications on their work machine whenever new articles arrive, but would want real-time updates at home. The set of feeds itself, on the other hand, would probably always be roamed.

To minimize the size of your roaming state, you might want to use the [Windows.Storage.Compression](#) API for file-based data. For this same reason, and because roaming state is app data, you should never use this mechanism to roam *user data*. Instead, use an online service like SkyDrive (or one of your own running, for example, on Windows Azure) to store user data in the cloud, and then roam URIs to those files as part of the roaming experience. More details on using SkyDrive through its REST API can be found on the [SkyDrive reference](#), on the [Skydrive core reference](#) (which includes a list of supported file types), and in the [PhotoSky sample](#). A backgrounder on this and other Windows Live services can also be found on the Building Windows 8 blog post entitled [Extending "Windows 8" apps to the cloud with SkyDrive](#).

By now you probably have a number of other questions forming in your mind about how roaming actually works: “How often is data synchronized?” “How do I manage different versions?” “What else should I know?” These are good questions, and fortunately there are good answers!

- Assuming there’s network connectivity, an app’s roaming state is roamed within 30 minutes on an active machine. It’s also roamed immediately when the user logs on or locks the machine. Locking the machine is always the best way to force a sync to the cloud. Note that if the cloud service is only aware of the user (that is, a Microsoft account) having only one device, synchronization with the cloud service happens only about once per day. When the service is aware that the user has multiple machines, it begins synchronizing within the 30-minute period.
- When saving roaming state, you can write values whenever you like, such as when those settings are changed. You don’t need to worry about writing settings as a group because Windows has a built-in debounce period to combine changes together and reduce overall network traffic.
- If you have a group of settings that really must be roamed together, manage these as a composite setting in your [roamingSettings](#) container.
- With files you create within the [roamingFolder](#), these will not be roamed so long as you have the file open for writing.
- Windows allows each app to have up to 8K worth of “high priority” settings that will be roamed within one minute, thereby allowing apps on multiple devices to stay much more closely in sync. To use this, create a single or composite setting within the root of your [roamingSettings](#) with the name *HighPriority*—that is,

`roamingSettings.values["HighPriority"]` (a *container* with this name will roam normally). So long as you keep the size of this setting below 8K, it will be roamed within a minute of being changed; if you exceed that size, it will be roamed with normal priority. See Scenario 9 of the Application Data sample for a demonstration.

- Systemwide user settings like the Start page configuration are automatically roamed apart from other apps. This also includes encrypted credentials saved by apps in the password vault; apps never need to attempt to roam passwords. Apps that create secondary tiles (as we'll see in Chapter 13, "Tiles, Notifications, the Lock Screen, and Background Tasks") can indicate whether such tiles should be copied to a new device when the app is installed.
- When there are multiple app data versions in use by the same app (with multiple app versions, of course), Windows will manage each version of the app data separately, meaning that newer app data won't be roamed to devices with apps that use older app data versions. In light of this, it's a good idea to not be too aggressive in versioning your app data since it will break the roaming connection between apps.
- The cloud service will retain multiple versions of roaming app data so long as there are multiple versions in use by the same Microsoft account. Only when all instances of the app have been updated will older versions of the roaming state be deleted.
- When an updated app encounters an older version of roaming state, it should load it according to the old version but save it as the new version and call `setVersionAsync`.
- Avoid using secondary versioning schemes within roaming state such that you introduce structural differences without changing the appdata version through `setVersionAsync`. Because the cloud service is managing the roaming state by this version number, and because the last writer always wins, some version of an app that expects to see some extra bit of data, and in fact saved it there, might find that it's been removed because a slightly older version of the app didn't write it. In short, it's best to avoid this.
- Even if all apps are uninstalled from a user's devices, the cloud service retains roaming data for "a reasonable time" (maybe 30 days) so that if a user reinstalls the app within that time period they'll find that their settings are still intact. To avoid this retention and explicitly clear roaming state from the cloud, use the `clearAsync` method.

Settings Pane and UI

We've now seen all the different APIs that an app can use to manage its state where storage is concerned, which is all you need for settings and other app data that are managed internally within the app. The question now is how to surface those settings that are user-configurable—for that we turn to

the Settings charm.

When the user invokes the Settings charm (which can also be done directly with the Win+i key), Windows displays the Settings pane, a piece of UI that is populated with various settings commands as well as system functions along the bottom. Apps can add their own commands, as most apps have some user-configurable settings of their own. Windows also guarantees that something always shows up for the app in this pane. It automatically displays the app name and publisher, a Rate and Review command that takes you to the Windows Store page for the app, an Update command if an update is available from the Store, and a Permissions command if the app has declared any capabilities in its manifest. (Note that Rate and Review won't appear for apps you run from Visual Studio since they weren't acquired from the Store.)

The Settings charm is always available no matter where you are in the app, so you don't need to think about having such a command on your app bar, nor do you ever need a settings command on your app canvas. That said, you can invoke the Settings charm programmatically, such as when you detect that a certain capability is turned off and you prompt the user about that condition. You might ask something like "Do you want to turn on geolocation for this app?" and if the user says Yes, you can invoke the Settings charm. This is done through the settings pane object returned from [Windows.UI.ApplicationSettings.SettingPane.getForCurrentView](#), whose `show` method display the UI (or throws a kindly exception if the app is in snapped view or doesn't have the focus, so don't invoke it under those conditions!). The `edge` property of the settings pane object also tells you if it's on the left or right side of the screen, depending on the left-to-right or right-to-left orientation of the system as a whole (a regional variance).

And with that we've covered all the methods and properties of this object! Yet the most interesting part is how we add our own commands to the settings pane. But let's first look at the guidelines for using Settings.

Design Guidelines for Settings

Beyond the commands that Windows automatically adds to the settings pane, the app can provide any number of others, typically around four. Because settings are global to an app, the commands you add are always the same: they are not sensitive to context. To say it another way, the *only* commands that should appear on the settings pane are those that are global to the app; commands that apply only to certain pages or contexts within a page should appear on the app bar or on the app canvas. Some examples of commands on the top-level settings pane are shown in Figure 8-2.

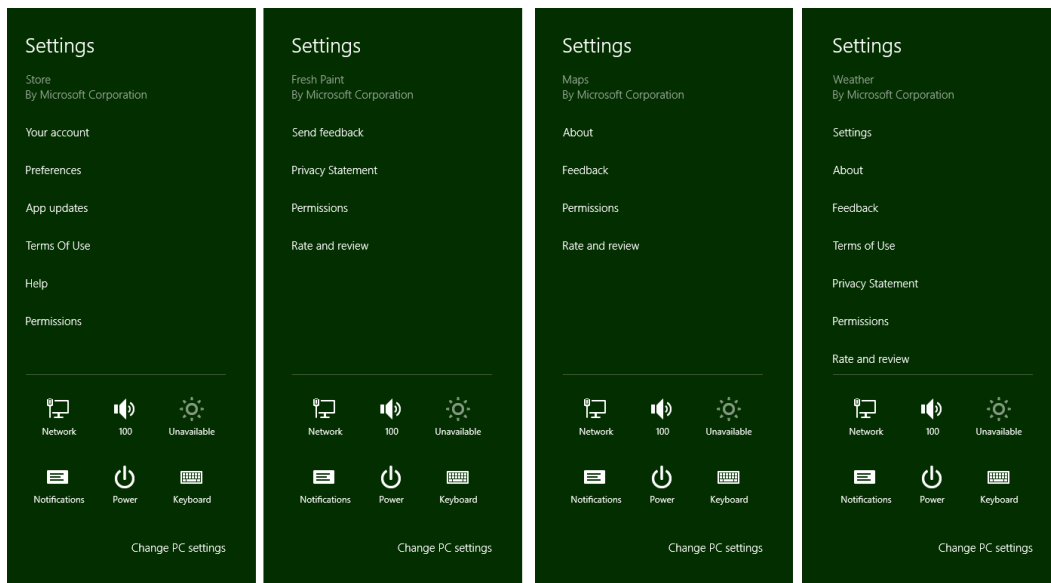


Figure 8-2 Examples of commands on the top-level settings pane. Notice that the lower section of the pane always has system settings and that the app name and publisher are always shown at the top. Permissions and Rate and Review are added automatically.

Each app-supplied command can do one of two things. First, a command can simply be a hyperlink to a web page. Some apps use links for their Help, Privacy Statement, Terms of Use, License Agreements, and so on, which will open the linked pages in a browser. The other option is to have the command invoke a secondary flyout panel with more specific settings controls or simply an `iframe` to display web-based content. You can provide Help, Terms of Use, and other textual content in both these ways rather than switch to the browser.

Note As stated in the [Windows 8 app certification requirements](#), section 4.1, apps that collect personal information in any way must have a privacy policy or statement. This must be included on the app's product description page in the Store as a minimum. Though not required, it is suggested that you also include a command for this in your Settings pane.

Secondary flyouts are created with the `WinJS.UI.SettingsFlyout` control as we'll see in a bit; some examples of such settings panes are shown in Figure 8-3. Notice that the secondary settings panes generally have two sizes: narrow (346px) and wide (646px). The design guidelines suggest that all secondary panes for an app are the same size—that is, don't make some narrow and some wide. You'll only have a couple of these panes anyway, so that shouldn't be a problem. Also note that the Permissions flyout, shown on the left of Figure 8-3, is provided by Windows automatically and is configured according to capabilities declared in your manifest. Some capabilities like geolocation are controlled in this pane; other capabilities are simply listed because the user is not allowed to turn them on or off. Access to the Internet as well as to various user data libraries are shown this way.

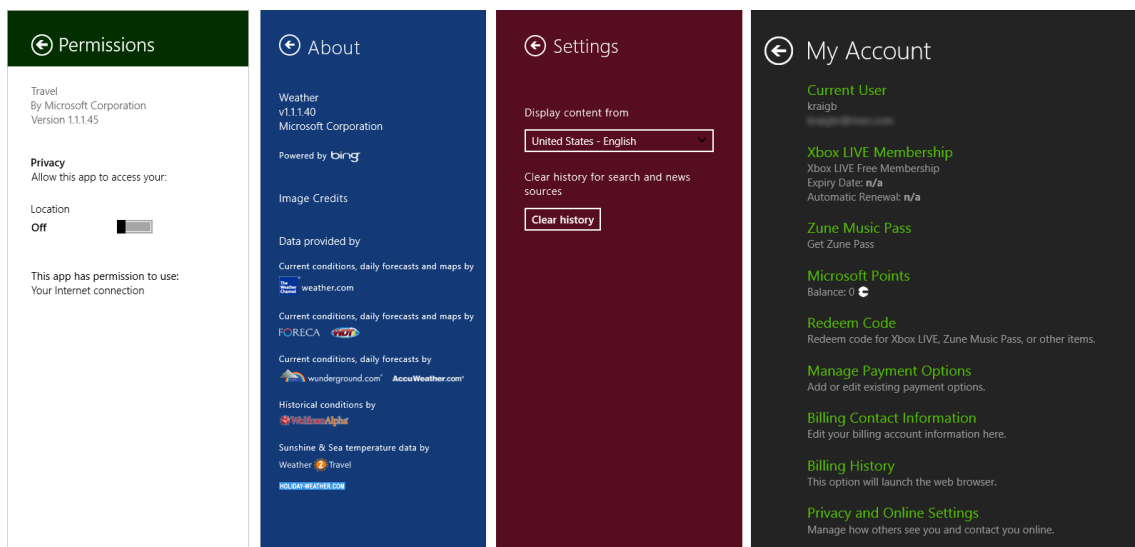


Figure 8-3 Examples of secondary settings panes in the Travel, Weather, News, and Music apps of Windows 8. The first three are the narrow size; the fourth is wide. Notice that each pane is branded appropriately for the app and provides a back button to return to the main Settings pane.

A common group of settings are those that allow the user to configure their roaming experience, which is to say, those settings that are roamed and those that are not. It is also recommended that you include account/profile management commands within Settings, as well as login/logout functionality. As noted in Chapter 7, logins and license agreements that are necessary to run the app at all should be shown upon launch. For ongoing login-related functions, and to review license agreements and such, create the necessary commands and panes within Settings. Refer to [Guidelines and checklist for login controls](#) for more information on this subject. Guidelines for a Help command can also be found on [Adding App Help](#).

Behaviorally, settings panes are light-dismiss but also have a header with a back button to return to the primary settings pane with all the commands. Because of the light-dismiss behavior, changing a setting on a pane applies the setting immediately: there is no OK or Apply button or other such UI. If the user wants to revert a change, she should just restore the original setting.

For this reason it's a good idea to use simple controls that are easy to switch back, rather than complex sets of controls that would be difficult to undo. The recommendation is to use toggle switches for on/off values (rather than check boxes), a button to apply an action (but without closing the settings UI), hyperlinks (to open the browser), text input boxes (which should be set to the appropriate type such as email address, password, etc.), radio buttons for groups of up to five mutually exclusive items, and a listbox (select) control for four to six text-only items.

In all your settings, think in terms of "less is more." Avoid having all kinds of different settings, because if the user is never going to find them, you probably don't need to surface them in the first place! Also, while a settings pane can scroll vertically, try to limit the overall size such that the user has

to pan down only once or twice.

Some other things to avoid with Settings:

- Don't use Settings for workflow-related commands. Those belong on the app bar or on the app canvas, as discussed in Chapter 7.
- Don't use a top-level command in the Settings pane to perform an action other than linking to another app (like the browser). That is, top-level commands should never execute an action *within* the app.
- Don't use settings commands to navigate within the app. Navigation is limited only to settings panes.
- Don't use `WinJS.UI.SettingsFlyout` as a general-purpose control.

And on that note, let's now look at the steps to use the control and Settings properly!

Populating Commands

The first part of working with Settings is to provide your specific commands when the Settings charm is invoked. Unlike app bar commands, these should always be the same no matter the state of the app; if you have context-sensitive settings, place commands for those in the app bar.

There are two ways to implement this process in a WinRT app written in HTML and JavaScript: using WinRT directly, or using the helpers in WinJS. Let's look at these in turn for a simple Help command.

To know when the charm is invoked through WinRT, obtain the settings pane object through `Windows.UI.ApplicationSettings.SettingsPane.getForCurrentView` and add a listener for its `commandsrequested` event:

```
// The n variable here is a convenient shorthand
var n = Windows.UI.ApplicationSettings;
var settingsPane = n.SettingsPane.getForCurrentView();
settingsPane.addEventListener("commandsrequested", onCommandsRequested);
```

Within your event handler, create `Windows.UI.ApplicationSettings.SettingsCommand` objects for each command, where each command has an `id`, a `label`, and an `invoked` function that's called when the command is invoked. These can all be specified in the constructor as shown below:

```
function onCommandsRequested(e) {
    // n is still the shortcut variable to Windows.UI.ApplicationSettings
    var commandHelp = new n.SettingsCommand("help", "Help", helpCommandInvoked);
    e.request.applicationCommands.append(commandHelp);
}
```

The second line of code is where you then add these commands to the settings pane itself. You do this by appending them to the `e.request.applicationCommands` object, which is something called a *vector* (a WinRT construct) of `SettingsCommand` objects that provides commands like `append` and `insertAt`. As you can see above, it's easy enough to append a command, and you'd make such a call

for each command or pass an array of such commands to the `replaceAll` methods instead of `append`. What then happens within the `invoked` handler for each command is the interesting part, and we'll come back to that in the next section.

You can also prepopulate the `applicationCommands` vector outside of the `commandsrequested` event; this is perfectly fine because your settings commands should be constant for the app. The [Quickstart: add app help](#) topic shows an example of this, which I've modified here to show the use of `replaceAll`:

```
var n = Windows.UI.ApplicationSettings;
var settingsPane = n.SettingsPane.getForCurrentView();
var vector = settingsPane.applicationCommands;

//Ensure no settings commands are currently specified in the settings charm
vector.clear();

var commands = [ new settingsSample.SettingsCommand("Custom.Help", "Help", OnHelp),
                  new n.SettingsCommand("Custom.Parameters", "Parameters", OnParameters)];
vector.replaceAll(commands);
```

This way, you don't actually need to register for or handle `commandsrequested` directly.

Now because most apps will likely use settings in some capacity, WinJS provides some shortcuts to this whole process. First, instead of listening for the WinRT event, simply assign a handler to `WinJS.Application.onsettings` (which is a wrapper for `commandsrequested`):

```
WinJS.Application.onsettings = function (e) {
    // ...
};
```

In your handler, create a JSON object describing your commands and store that object in the event object, specifically `e.detail.applicationcommands`. Mind you, this is *different* from the WinRT object—just setting this property accomplishes nothing. What comes next is passing the now-modified event object to `WinJS.UI.SettingsFlyout.populateSettings` like so (taken from Scenario 2 of the [App Settings sample](#)):

```
WinJS.Application.onsettings = function (e) {
    e.detail.applicationcommands =
        { "help": { title: "Help", href: "/html/2-SettingsFlyout-Help.html" } };
    WinJS.UI.SettingsFlyout.populateSettings(e);
};
```

The `populateSettings` method walks the `e.details.applicationcommands` object and call the WinRT `applicationCommands.append` method for each item. This gives you a more compact method to accomplish what you'd do with WinRT, and it also simplifies the implementation of settings commands, as we'll see next.

Implementing Commands: Links and Settings Flyouts

Technically speaking, within the `invoked` function you assign to any command you can really do

anything. Truly! Of course, as described in the design guidelines earlier, there are recommendations for how to use settings and how not to use them. For example, settings commands shouldn't act like app bar commands that affect content, nor should they navigate within the app itself. Ideally, a settings command does one of two things: either launch a hyperlink (to open a browser) or display a secondary settings pane.

In the base WinRT model for settings, launching a hyperlink uses the [Windows.System.Launcher.-launchUriAsync](#) API as follows:

```
function helpCommandInvoked(e) {  
    var uri = new Windows.Foundation.Uri("http://example.domain.com/help.html");  
    Windows.System.Launcher.launchUriAsync(uri).done();  
}
```

In the second case, secondary panes are implemented with the [WinJS.UI.SettingsFlyout](#) control. Again, technically speaking, you're not required to use this control: you can display any UI you want within the `invoked` handler. The `SettingsFlyout` control, however, provides for the recommended narrow and wide sizes, supplies enter and exit animations, fires animations like `[before | after][show | hide]`⁴¹ and other such features. And since you can place any HTML you want within the control, including other controls, and the flyout will automatically handle vertical scrolling, there's really no reason *not* to use it.

As a WinJS control, you can declare a `SettingsFlyout` for each one of your commands in markup (making sure `WinJS.UI.process/processAll` is called, which handles any other controls in the flyout). For example, Scenario 2 of the [App Settings sample](#) has following flyout for help (omitting the text content and reformatting a bit), shown in Figure 8-4:

```
<div data-win-control="WinJS.UI.SettingsFlyout" aria-label="Help settings flyout"  
    data-win-options="{settingsCommandId:'help', width:'wide'}">  
    <!-- Use either 'win-ui-light' or 'win-ui-dark' depending on the contrast between  
         the header title and background color; background color reflects app's personality -->  
    <div class="win-ui-dark win-header" style="background-color:#00b2f0">  
        <button type="button" onclick="WinJS.UI.SettingsFlyout.show()" class="win-backbutton"></button>  
        <div class="win-label">Help</div>  
          
    </div>  
    <div class="win-content ">  
        <div class="win-settings-section">  
            <h3>Settings charm usage guidelines summary</h3>  
            <!-- Other content omitted -->  
        </div>  
    </div>  
</div>
```

⁴¹ How's that for a terse combination of four event names?

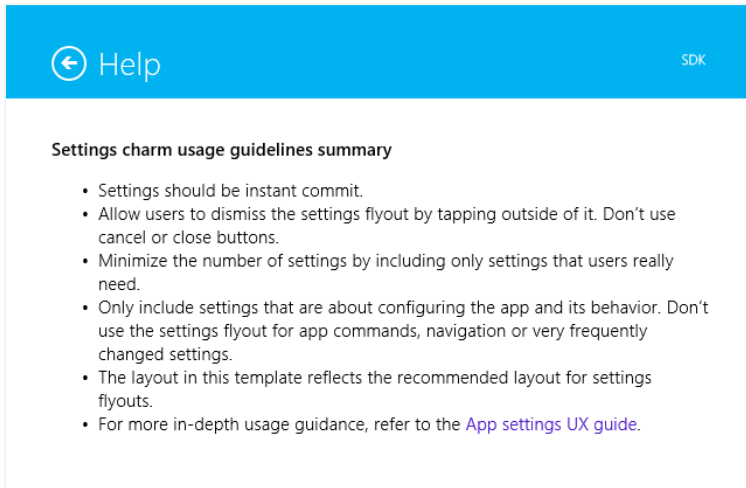


Figure 8-4 The Help settings flyout (truncated vertically) from Scenario 2 of the App Settings sample. Notice the hyperlink on the lower right.

As always, there are options for this control as well as a few applicable `win-*` style classes. The only two options are `settingsCommandId`, for obvious purpose, and `width`, which can be `'narrow'` or `'wide'`. We see these both in the example above. The styles that apply here are `win-settingsflyout`, which styles the whole control (typically not used except for scoping other style rules), and `win-ui-light` and `win-ui-dark`, which apply a light or dark theme to the parts of the flyout. In this example, we use the dark theme for the header while the rest of the flyout uses the default light theme.

In any case, you can see that everything within the control is just markup for the flyout contents, nothing more, and you can wire up events to controls in the markup or in code. One bit I omitted from the example is that you're also free to use hyperlinks here, such as to launch the browser to open a fuller Help page. You can also use an `iframe` to directly host web content within a settings flyout—that's no problem at all, and is, in fact, demonstrated in Scenario 3 of the same sample.

So how do we get this flyout to show when a command is invoked on the top-level settings pane? The easy way is to let WinJS take care of the details using the information you provide to `WinJS.UI.SettingsFlyout.populateSettings`. Here's the example again from Scenario 2, as we saw in the previous section:

```
WinJS.Application.onSettings = function (e) {
    e.detail.applicationCommands =
        { "help": { title: "Help", href: "/html/2-SettingsFlyout-Help.html" } };
    WinJS.UI.SettingsFlyout.populateSettings(e);
};
```

In the JSON you assign to `applicationCommands`, each object identifies both a command and its associated flyout. The name of the object is the flyout id ("help"), its `title` property provides the command label for the top-level settings pane ("Help" in the above), and its `href` property identifies

the HTML page where the flyout with that id is declared ("/html/2-SettingsFlyout-Help.html").

With this information, WinJS can both populate the top-level settings pane and provide automatic invocation of the desired flyout (calling `WinJS.UI.process` all along the way) without you having to write any other code. This is why in most of the scenarios of the sample you don't see any explicit calls to `showSettings`, just a call to `populateSettings`.

Note The `href` property used with `populateSettings` must always refer to in-package content; it *cannot* be used to create settings commands that launch a URI. For that you need to use your own command handlers, as shown at the beginning of this section.

Programmatically Invoking Settings Flyouts

Let's now see what's going on under the covers. In addition to being a control that you use to define a specific flyout, `WinJS.UI.SettingsFlyout` has a couple of other static methods: `show` and `showSettings` (in addition to the static `populateSettings`). The `show` method specifically brings out the top-level Windows settings pane—that is, `Windows.UI.ApplicationSettings.SettingsPane`. This is why you see the back button's `click` event in the above markup wired directly to `show`, because the back button should return to that top-level UI.

The `showSettings` method, on the other hand, shows a *specific* settings flyout that you define somewhere in your app. The signature of the method is `showSettings(<id> [, <page>)` where `<id>` identifies the flyout you're looking for and the optional `<page>` parameter identifies an HTML document to look in if a flyout with `<id>` isn't found in the current document. That is, `showSettings` will always start by looking in the current `document` for a `WinJS.UI.SettingsFlyout` element that has a matching `settingsCommandId` property or a matching HTML `id` attribute. If such a flyout is found, that UI is shown.

If the markup above was contained in the same HTML page that's currently loaded in the app, the following line of code will show that flyout:

```
WinJS.UI.SettingsFlyout.showSettings("help");
```

In this case you could also omit the `href` part of the JSON object passed to `populateCommands`, but only again if the flyout is contained within the current HTML document already.

The `<path>` parameter, for its part, allows you to separate your settings flyouts from the rest of your markup; its value is a relative URI within your app package. The App Settings sample uses this to place the flyout for each scenario into a separate HTML file. You can also place all your flyouts in one HTML file, so long as they have unique ids. Either way, if you provide a `<path>`, `showSettings` will load that HTML into the current page using `WinJS.UI.Pages.load` (which calls `WinJS.UI.processAll`), scans that DOM tree for a matching flyout with the given `<id>`, and shows it. Failure to locate the flyout will cause an exception.

Scenario 5 of the sample shows this form of programmatic invocation, this is also a good example (see Figure 8-5) of a vertically scrolling flyout:


```
WinJS.UI.SettingsFlyout.showSettings("defaults", "/html/5-SettingsFlyout-Settings.html");
```

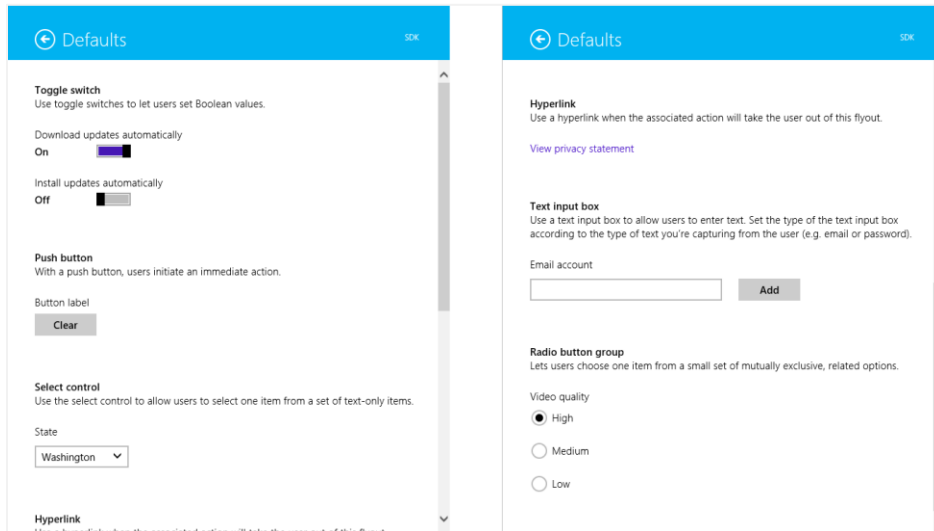


Figure 8-5 The settings flyout from Scenario 5 of the App Settings sample, showing how a flyout supports vertical scrolling; note the scrollbar positions for the top portion (left) and the bottom portion (right).

A call to `showSettings` is thus exactly what you use within any particular command's `invoked` handler and is what WinJS sets up within `populateCommands`. But it also means you can call `showSettings` from anywhere else in your code when you want to display a particular settings pane. (Also see the sidebar on permissions below.) For example, if you encounter an error condition in the app that could be rectified by changing a setting, you can provide a button in the message dialog of notification flyout that calls `showSettings` to open that particular pane. And for what it's worth, the `hide` method of that flyout will dismiss it; it doesn't affect the top-level settings pane for which you must use `Windows.UI.ApplicationSettings.SettingsPane.getForCurrentView.hide`.

You might use `showSettings` and `hide` together, in fact, if you need to navigate to a third-level settings pane. That is, one of your own settings flyouts could contain a command that calls `hide` on the current flyout and then calls `showSettings` to invoke another. The back button of that subsidiary flyout (and it should always have a back button) would similarly call `hide` on the current flyout and `showSettings` to make its second-level parent reappear. That said, we don't recommend making your settings so complex that third-level flyouts are necessary, but the capability is there if you have a particular scenario that demands it.

Knowing how `showSettings` tries to find a flyout is also helpful if for some reason you want to create a `WinJS.UI.SettingsFlyout` programmatically. So long as such a control is in the DOM when you call `showSettings` with its id, WinJS will be able to find it and display it like any other. It would also work, though I haven't tried this and it's not in the sample, to use a kind of hybrid approach. Because `showSettings` loads the HTML page you specify as a page control with `WinJS.UI.Pages.load`, that page can also include its own script wherein you define a page control object with methods like

`processed` and `ready`. Within those methods you could then make specific customizations to the settings flyout defined in the markup.

Sidebar: Changes to Permissions

Although you can programmatically invoke any app-specific settings flyouts, the same is not true for those settings commands that are provided by Windows, such as Permissions. This is why the recommendation is to simply display an error message to tell the user to invoke Permissions rather than just opening that UI directly.

A common question along these lines is whether an app can receive events when the user changes settings within the Permissions pane. The answer is unfortunately no, which means that you discover whether access is disallowed only by handling Access Denied exceptions when you try to use the capability. To be fair, though, you always have to handle denial of a capability gracefully because the user can always deny access the first time you use the API. When that happens, you again display a message about the disabled permission (as shown with the Here My Am! app from Chapter 7) and provide some UI to reattempt the operation. But the user still needs to invoke the Permissions settings manually. (Refer also to the [Guidelines for devices that access personal data](#) for more details.)

User Data: Libraries, File Pickers, and File Queries

Now that we've thoroughly explored app data and app settings, we're ready to look at the other part of state: user data. User data, again, is all the good stuff an app might use or generate that isn't specifically tied to the app. Multiple apps might be able to work with the same files, such as pictures and music, and user data always stays on a device regardless of what apps are present.

Our first concern with user data is where to put it and where to access it, which involves the various user data libraries, removable storage, and the file pickers. Using the access cache is also important to remember the fact that a user once granted access to a file or folder that we're normally not allowed to touch programmatically. The good thing about all such files and folders is that working with them happens through the same `StorageFolder` and `StorageFile` classes we've already seen. The other main topic we'll explore is that of file queries, a richer way to enumerate the contents of folders and libraries that lend very well to visual representations within controls like a `ListView`.

As we've seen, a WinRT app, by default, has access only to its package and its AppData folders. This means that, by default, it doesn't actually have any access to typical locations for user data! There are then two ways that such access happens:

- Declare a library capability in the manifest.
- Let the user choose a location through the File Picker.

We'll look first at the File Picker, because in many cases it's all you really need in an app and you don't need to declare any capabilities at all! But there are other scenarios—such as gallery-style apps—where you need direct access, so there are five capabilities in the manifest for this purpose, as shown in Figure 8-6 (left side). Three of them—Music Library, Pictures Library, and Videos Library—grant full read-write access to the user's Music, Pictures, and Videos folders. These appear on the app's product page in the Windows Store and on the Permissions settings pane, but they are not subject to user consent at run time. Of course, if it's not obvious why you're declaring these capabilities, be sure to explain yourself on your product page. And as for Documents Library and Removable Storage, simply declaring the capability isn't sufficient: you also need to declare specific file type associations to which you're then limited.

Sidebar: The Background Transfer API

A topic that's relevant to user data, but one that we won't cover in detail until Chapter 14, "Networking," is the [Windows.Networking.BackgroundTransfer](#) API of WinRT. This API allows you to run downloads and uploads independently of app lifetime—that is, while the app is running, suspended, or not running at all. This API is provided because transfer of large files to and from online resources is a common need for apps but one that doesn't really need the apps themselves to run in the background and consume power. Instead, apps set up transfer operations with the system that will continue if the app is shut down. When the app is relaunched, it can then check on the status of those transfers.

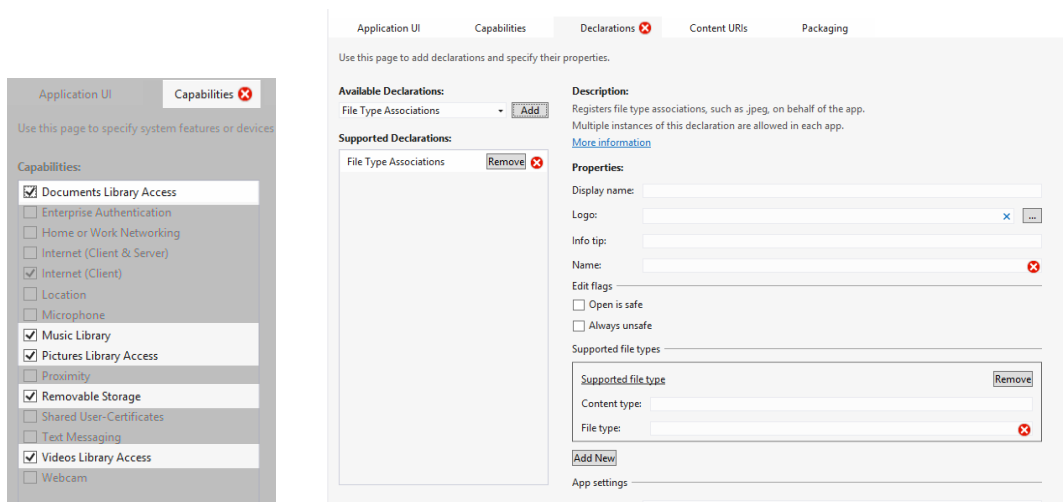


Figure 8-6 Capabilities related to user data in the manifest editor (left) and the file type association editor (right). Notice that the red X appears on Capabilities when additional declarations are needed in conjunction with this capability. The red X on Declarations indicates that the information is not yet complete.

Using the File Picker

Although the File Picker doesn't sound all that glamorous, it's actually, to my mind, one of the coolest

features in Windows 8. “Wait a minute!” you say, “How can a UI to pick a file or folder be, well, *cool!*” The reason is that this is *the* place where the users can browse and select from their entire world of data. That world includes not only what’s on their local file system or the local network, but also any data that’s made available by what are called *file picker providers*. These are apps that specifically take a library of data that’s otherwise buried behind a web service or within an app’s own database and makes it appear as if it’s part of the local file system.

Think about this for a moment (as I invited you to do way back in Chapter 1). When you want to work with an image from a photo service like Flickr or Picasa, for example, what do you typically have to do? First step is to download that file to the local file system within some app that gives you an interface to that service (which might be a web app). Then you can make whatever edits and modifications you want, after which you typically need to upload the file back to the service. Well, that’s not so bad, except that it’s time consuming, forces you to switch between multiple apps, and eventually litters your system with a bunch of temporary files, the relationship of which to your online files is quickly forgotten.

Having a file picker provider that can surface such data directly, both for reading and writing, eliminates all those intermediate steps, along with the need to switch apps. Such a provider over a photo service, for example, simply makes that online library appear local, allowing other apps to load them, edit them, and save them just like consumers are already accustomed to doing but without all the overhead and without having to leave the app they were originally using. Consuming apps, then, don’t need to know anything about those other services, and they automatically have access to more services as more provider apps are installed. What’s more, providers can also make data that isn’t normally stored as files appear as though they are. For example, the Windows 8 Camera app is a file picker provider, where through that contract you can activate your camera, take a picture, and have it returned as if you loaded it from the file system. All of this gives users a very natural means to flow in and out of data no matter where it’s stored. Like I said, I think this is a very cool feature!

We’ll look more at the question of providers in Chapter 12. Our more immediate concern is how we make use of these file pickers. This goes back to the earlier question about how you work with files in WinRT, the answer to which is that you had to start with some API that gives you a [StorageFile](#) or [StorageFolder](#) object. And again, the file picker is just such an API.

The File Picker UI

Before looking at the code, let’s familiarize ourselves with the file picker UI itself. When invoked, you’ll see a full-screen view like that in Figure 8-7, which shows the picker in single-selection mode with a “thumbnail” view. In such a view, items are shown as images in a `ListView`, with a rich tooltip control appearing when you hover over an item (and I did change my color scheme now to a rich green rather than the dark copper). In a way, the file picker itself is like an app that’s invoked for this purpose, and it’s designed to be beautiful and immersive just like other WinRT apps.

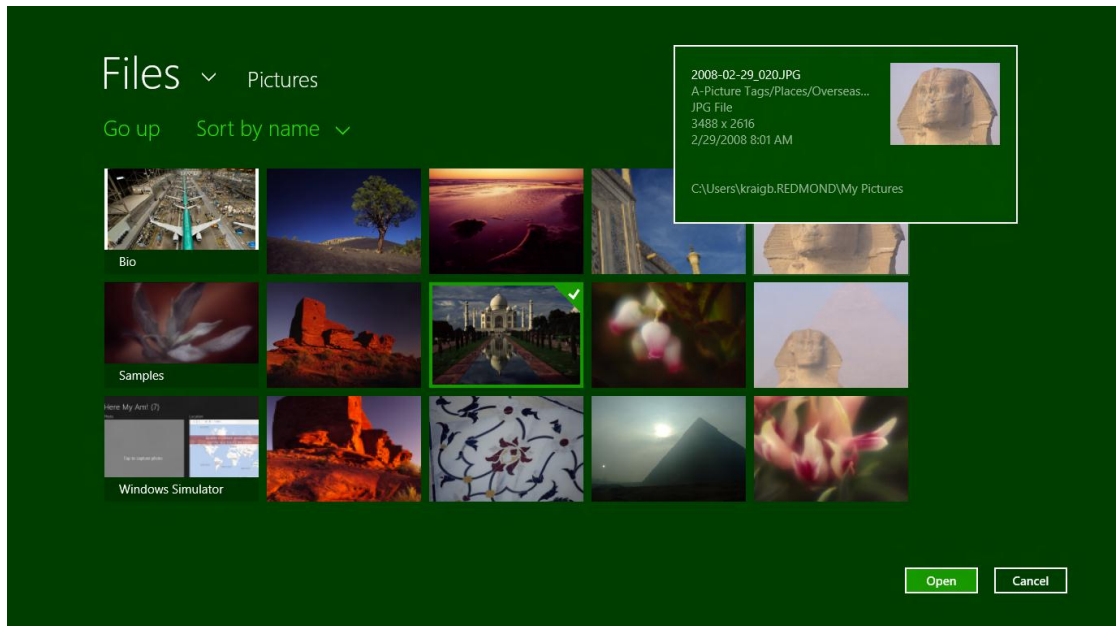


Figure 8-7 A single-selection file picker on the Pictures library in thumbnail view mode, with a hover tooltip showing for one of the items (the head of the Sphinx) and the selection frame showing on another (the Taj Mahal).

In Figure 8-7, the Pictures heading shows the current location of the picker. The Sort By Name drop-down list lets you choose other sorting criteria, and the drop-down list next to the Files header lets you choose other locations, as shown in Figure 8-8. These locations include other areas of the file system (though never protected areas like the Windows folder or Program Files), network locations, *and* other provider apps. Note that app options are not provided in the folder picker—just the file picker.

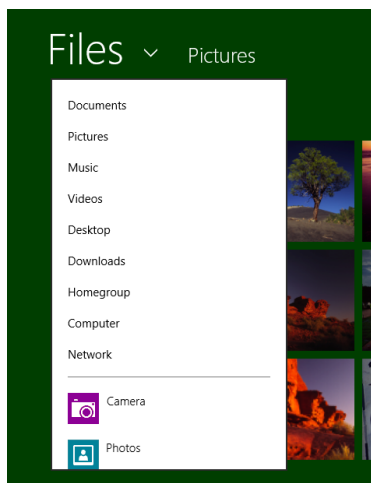


Figure 8-8 Selecting other locations in which to browse files; notice that apps are listed along with file system locations.

Choosing another file system location navigates there, of course, from which you can browse into other folders. Selecting an app, on the other hand, launches that app through the file picker provider contract, in response to which it configures itself to serve in that capacity like the Camera app in Figure 8-9, even to the point where the drop-down list next to the heading lets you easily switch back to other picker locations. In short, a provider app really is just an extension to the File Picker UI, but a very powerful one at that. And ultimately such an app just returns an appropriate `StorageFile` object that makes its way back to the original app. It's quite a lot happening with just a single call to the file picker API!

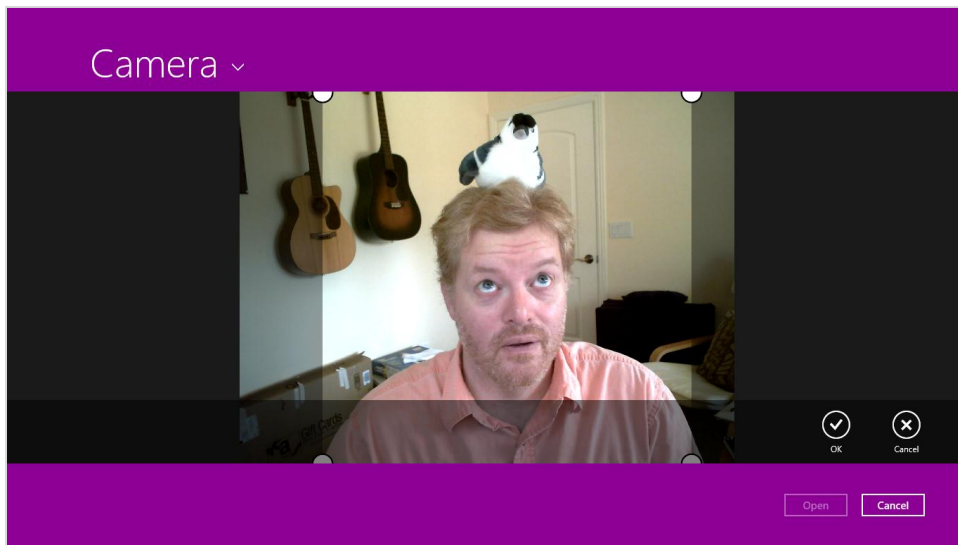


Figure 8-9 The camera app invoked through the file picker provider contract. Where did that nuthatch come from?

The file picker has a couple of other modes. One is the ability to select multiple files—even from different apps!—as shown in Figure 8-10, where all the selections are placed into what's called the *basket* on the bottom of the screen. The picker can also be used to select a folder, as shown in Figure 8-11, or a save location and filename, as shown in Figure 8-12.

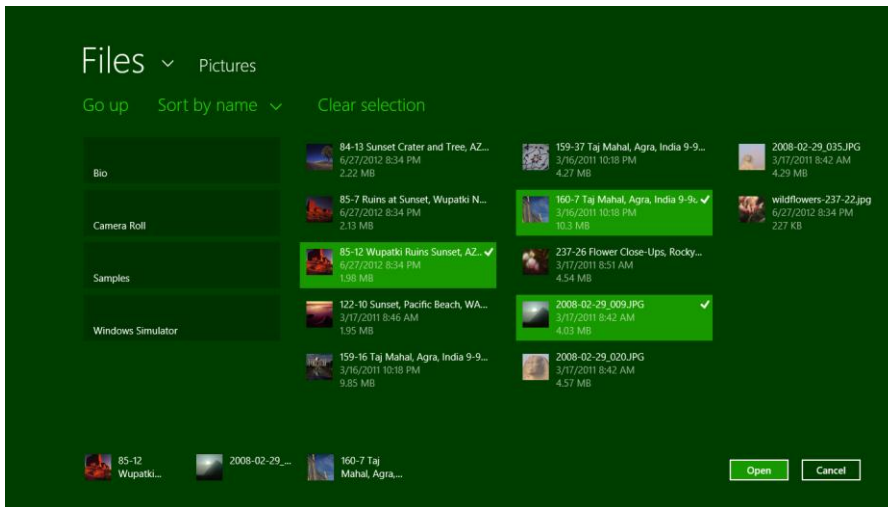


Figure 8-10 The file picker in multiselect mode with the selection basket at the bottom. What shown here is also the “list” view mode that’s set independently from the selection mode.

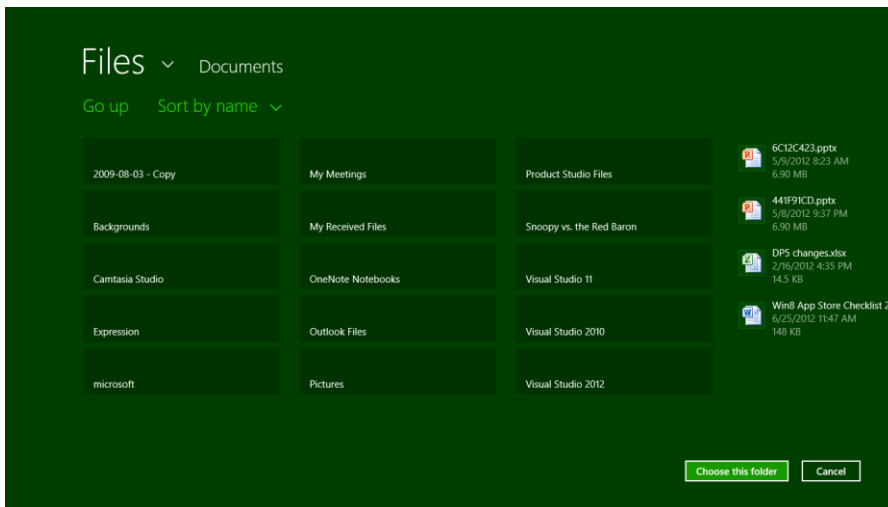


Figure 8-11 The file picker used to select a folder.

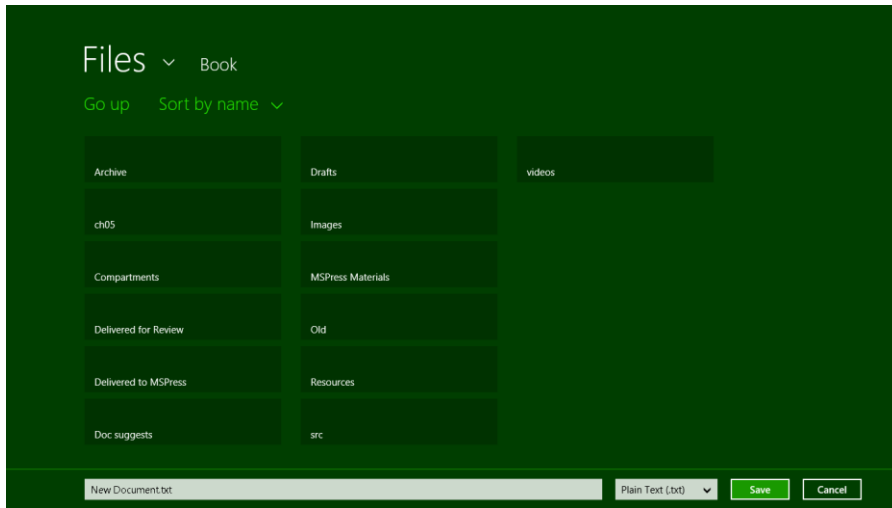


Figure 8-12 The file picker used to select a save location and filename.

The File Picker API (and a Few Friends)

Now that we've seen the visual results of the file picker, let's see how we invoke it from our app code through the API in [Windows.Storage.Pickers](#). All the images we just saw came from the [Access and save files using the file picker sample](#) (love the name!), so we'll also use that as the source of our code.

For starters, Scenario 1 in its `pickSinglePhoto` function (`scenario1.js`) shows how to use the picker to obtain a single `StorageFile` for opening (which implies both reading and writing, as picking a save location implies a Save As to change the filename):

```
function pickSinglePhoto() {
    // Verify that we are currently not snapped, or that we can unsnap to open the picker
    var currentState = Windows.UI.ViewManagement.ApplicationView.value;
    if (currentState === Windows.UI.ViewManagement.ApplicationViewState.snapped &&
        !Windows.UI.ViewManagement.ApplicationView.tryUnsnap()) {
        // Fail silently if we can't unsnap
        return;
    }

    // Create the picker object and set options
    var openPicker = new Windows.Storage.Pickers.FileOpenPicker();
    openPicker.viewMode = Windows.Storage.Pickers.PickerViewMode.thumbnail;
    openPicker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.picturesLibrary;

    // Users expect to have a filtered view of their folders depending on the scenario.
    // For example, when choosing a documents folder, restrict the filetypes to documents
    // for your application.
    openPicker.fileTypeFilter.replaceAll([".png", ".jpg", ".jpeg"]);

    // Open the picker for the user to pick a file
    openPicker.pickSingleFileAsync().done(function (file) {
        if (file) {
```



```

        // Application now has read/write access to the picked file
    } else {
        // The picker was dismissed with no selected file
    }
    });
}

```

As you can see, you should not try to invoke the File Picker when in snapped view; this will, like the Settings Pane, cause an exception. You can check for such a condition ahead of time, as shown here, or you can add an error handler within the `done` at the end.⁴² In any case, to invoke the picker we create an instance of `Windows.Storage.Pickers.FileOpenPicker`, configure it and then call its `pickSingleFileAsync` method. The result of `pickSingleFileAsync` is the `file` argument given to the completed handler, which will be either a `StorageFile` object for the selected file or `null` if the user canceled.

With the configuration, here we're setting the picker's `viewMode` to `thumbnail` (from the enumeration `Windows.Storage.Pickers.PickerViewMode`), resulting in the view of Figure 8-7. The other possibility here is `list`, which gives a view like Figure 8-10.

We also set the `suggestedStartLocation` to the `picturesLibrary`, which is a value from the `Windows.Storage.Pickers.PickerLocationId` enumeration; other possibilities are `documentsLibrary`, `computerFolder`, `desktop`, `downloads`, `homeGroup`, `musicLibrary`, and `videosLibrary`, basically all the other locations you see in Figure 8-8. Note that using these locations does *not* require you to declare capabilities in your manifest because by using the picker, the user is giving consent for you to access those files. If you check the manifest in this sample, you'll see that no capabilities are declared at all.

The one other property we set is the `fileTypeFilter` (a `FileExtensionVector` object) to indicate the type of files we're interested in, which are PNG and JPEG files in this case. Beyond that, the `FileOpenPicker` also has a `commitButtonText` property, which controls the label of the primary button in the UI (the one that's not Cancel), and `settingsIdentifier`, a means to essentially remember different contexts of the file picker. For example, an app might use one identifier for selecting pictures, where the starting location is set to the pictures library and the view mode to thumbnails, and another id for selecting documents with a different location and perhaps a list view mode.

This sample, as you can also see, doesn't actually do anything with the file once it's obtained, but it's quite easy to see what we might do. We can, for instance, simply pass the `StorageFile` to `URL.createBlobURL` and assign the result to an `img.src` property to display the contents of that file. The same thing could be done with audio and video as well, possibilities that are all demonstrated in Scenario 1 of the [Using a blob to save and load content sample](#) I mentioned earlier in this chapter. That same sample also shows reading the file contents through the HTML `FileReader` API, something that you could also do with other WinRT and WinJS APIs, as we've seen. You could also transcode an

⁴² The sample, it should be noted, uses `then` instead of `done` on that last async call; while `then` works, it should actually be `done` especially if you're going to handle exceptions there.

image (or other media) in the [StorageFile](#) to another format (as we'll see in Chapter 10), retrieve thumbnails as shown in the [Retrieve thumbnails for files and folders sample](#), or use the [StorageFile](#) methods to make a copy in another location, rename the file, and so forth. But from the file picker's point of view, its particular job was well done!

Returning now to the file picker sample, picking multiple files is pretty much the same story as shown in the [pickMultipleFiles](#) function of `scenario2.js`, except that here we're using the `list` view mode and starting off in the `documentsLibrary`. Again, these start locations don't require capability declarations in the manifest:⁴³

```
function pickMultipleFiles() {
    // Verify that we are currently not snapped, etc... (some code omitted)

    // Create the picker object and set options
    var openPicker = new Windows.Storage.Pickers.FileOpenPicker();
    openPicker.viewMode = Windows.Storage.Pickers.PickerViewMode.list;
    openPicker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.documentsLibrary;
    openPicker.fileTypeFilter.replaceAll(["*"]);

    // Open the picker for the user to pick a file
    openPicker.pickMultipleFilesAsync().done(function (files) {
        if (files.size > 0) {
            // Application now has read/write access to the picked file(s)
        } else {
            // The picker was dismissed with no selected file
        }
    });
}
```

When picking multiple files, the result of [pickMultipleFilesAsync](#) is a [FilePickerSelectedFilesArray](#) object, which you can access like any other array using `[]` (though it has limited methods otherwise).

Scenario 3 of the sample shows a call to [pickSingleFolderAsync](#) (see the [pickFolder](#) function in `scenario3.js`), where the result of the operation is a [StorageFolder](#). Here you must indicate a [fileTypeFilter](#) that's relevant to the purpose of picking a folder; this helps users pick an appropriate location in which to enumerate files or perhaps create new ones:

```
function pickFolder() {
    // Verify that we are currently not snapped... (some code omitted)

    // Create the picker object and set options
    var folderPicker = new Windows.Storage.Pickers.FolderPicker;
    folderPicker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.desktop;
    folderPicker.fileTypeFilter.replaceAll([".docx", ".xlsx", ".pptx"]);

    folderPicker.pickSingleFolderAsync().then(function (folder) {
        if (folder) {
```

⁴³ Again, the actual sample uses `then` instead of `done` on the async call; I show `done` here which is the better choice.

```

        // Cache folder so the contents can be accessed at a later time
        Windows.Storage.AccessCache.StorageApplicationPermissions.futureAccessList
            .addOrReplace("PickedFolderToken", folder);
    } else {
        // The picker was dismissed with no selected file
    }
    });
}

```

In this example we also see how to save that selected [StorageFolder](#) in the [Windows.Storage.-AccessCache](#) for future use. Again, by selecting this folder the user has granted the app programmatic access to its contents. However, that access is limited to the current session. To maintain that access, the app must save the storage item in the [futureAccessList](#) of the cache, where it can be later retrieved using the [futureAccessList.getFolderAsync](#), [getItemAsync](#), or [getFileAsync](#) methods (where access to files obtained from another app could trigger an update from a source, if the app supports that cached file updating, as we'll see in Chapter 12). As before, refer to Scenario 6 of the [File Access sample](#) for more on this feature, and note that the [AccessCache](#) API also provides for recently used items as well.

For the final file picker use case, Scenario 4 of the file picker sample creates a [FileSavePicker](#) object and calls its [pickSaveFileAsync](#) method, resulting in the UI of Figure 8-12:

```

function saveFile() {
    // Verify that we are currently not snapped... (some code omitted)

    // Create the picker object and set options
    var savePicker = new Windows.Storage.Pickers.FileSavePicker();
    savePicker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.documentsLibrary;
    // Dropdown of file types the user can save the file as
    savePicker.fileTypeChoices.insert("Plain Text", [".txt"]);
    // Default file name if the user does not type one in or select a file to replace
    savePicker.suggestedFileName = "New Document";

    savePicker.pickSaveFileAsync().done(function (file) {
        if (file) {
            // Prevent updates to the remote version of the file until we finish making changes
            // and call CompleteUpdatesAsync.
            Windows.Storage.CachedFileManager.deferUpdates(file);

            // write to file
            Windows.Storage.FileIO.writeTextAsync(file, file.name).done(function () {
                // Let Windows know that we're finished changing the file so the other app
                // can update the remote version of the file.
                // Completing updates might require Windows to ask for user input.
                Windows.Storage.CachedFileManager.completeUpdatesAsync(file)
                    .done(function (updateStatus) {
                        if (updateStatus === Windows.Storage.Provider.FileUpdateStatus.complete) {
                        } else {
                            // ...
                        }
                    })
                }
            });
        }
    });
}

```

```

        });
    } else {
        // The picker was dismissed
    }
});
}

```

The `FileSavePicker` has many of the same properties as the `FileOpenPicker`, but it replaces `fileTypeFilter` with `fileTypeChoices` (to populate the drop-down list) and includes a `suggestedFileName` (a string), `suggestedSaveFile` (a `StorageFile`), and `defaultFileExtension` (a string). What's interesting in the code above is that once a file is saved, we use the [Windows.Storage.CachedFileManager](#) to defer updates to the file we're saving until all our async write operations are complete. This is done, as the comments suggest, to make sure that the file picker provider doesn't try to update the data on a service or other remote resource until we've actually written what we need. And what you see here with the `CachedFileManager` is all there is to it: just use this pattern any time you save a file obtained from a file picker because it might be coming from a remote source. The `CachedFileManager` will take care of notifying the source as necessary.

You should also use this same pattern when you save a file obtained from the access cache as well, because it might have come from a file picker originally. You wouldn't, on the other hand, employ this pattern for files that you know are local and always accessible, like those in your AppData folders.

Media Libraries

Now that we've seen understood the capabilities of the file picker, we can turn our attention to the other libraries. But before you start checking off capabilities in your manifest, pause for a moment to ask this: are those capabilities actually needed? The file pickers provide very extensive access to all these libraries without needing those capabilities at all. Through the pickers you can have the user select one or more files to open, manipulate, and save; the user can indicate a folder to which you'll then have access; and the user can indicate a new filename in which to save user data.

You only need specific library access, then, if you're going to work within any of these libraries without using the file picker. For example, if you want to enumerate the contents of the Pictures or Music folder to create a data source for a ListView or FlipView control, as we did in Chapter 5, you do need to declare the capability. The `WinJS.UI.StorageDataSource` object we used back then does exactly this.

We can be even more specific. Without going through the file picker, there is only *one* way to gain programmatic access to the media libraries if you've declared the capability: obtain a `StorageFolder` from the [Windows.Storage.KnownFolders](#) object (the `StorageDataSource` uses this internally). For media, the applicable properties here are `picturesLibrary`, `musicLibrary`, and `videosLibrary`. Without the appropriate capability, trying to retrieve these will throw an access denied exception.

In short, if you don't find a need to use access `KnownFolders`, you don't need to declare the capabilities! And remember that since all your declared capabilities are listed for your app in the Windows Store and might make consumers think twice about installing your app, fewer is definitely

better.

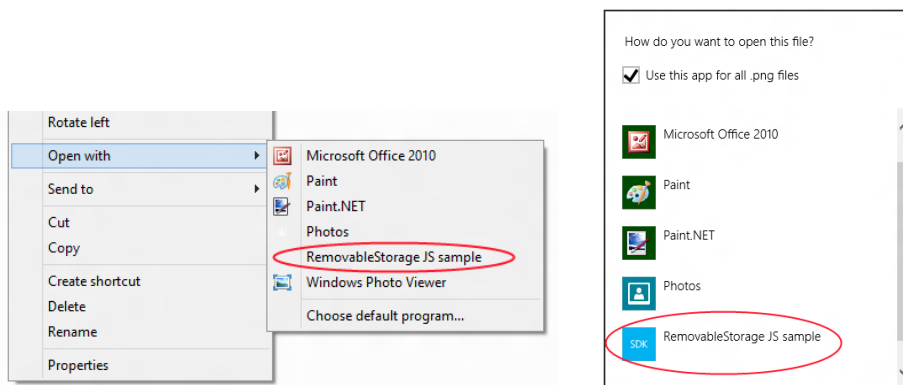
Either way, working with media libraries involves [StorageFolder](#) and [StorageFile](#) objects pretty much like any other storage location. One difference, however, is that you can work with the metadata often included with media files; we'll see a little more of this in Chapter 10.

Documents and Removable Storage

As with the media libraries, access to the user's documents folder as well as removable storage devices are controlled by capability declarations, and aside from the file picker the only way to get an appropriate [StorageFolder](#) is through [Windows.Storage.KnownFolders](#) (specifically, [documentsLibrary](#) and [removableDevices](#)). In the latter case, you get a [StorageFolder](#) that contains a subfolder for each removable device.

Again, before you start working on your manifest, ask yourself if you really need open access to these folders or whether the file pickers are entirely sufficient. If you can't think of why, exactly, you'd access [KnownFolders](#), then you don't actually need to declare the capability. I say this because developers have, in the past, mistakenly assumed that they needed [documentsLibrary](#) access just to save and load arbitrary user data, but that's not actually true. It's really needed only if you're going to create your own file-browsing UI or otherwise enumerate folder contents. In fact, when you try to upload an app to the Windows Store that declared documents library access, you'll be asked to indicate exactly how it's being used.

Even when you do declare the capabilities for documents and removable devices, access is yet restricted to specific file types you declare in the Declarations section of your manifest. And here's the rub: by declaring associations for those file types, you are also declaring that your app is available to service files of that type at any time. For example, the [Removable storage sample](#) in the SDK, in order to demonstrate access to removable devices, declares associations with .gif, .jpg, and .png files. As a result, it shows up in Open With lists like the context menu of Windows Explorer on the desktop and the default program selector:



The same is also true for documents (see the [File Access sample](#) again), so unless your app is really

positioned to service those file types, you probably don't need these capabilities. After all, a user can grant you access to a whole folder through the folder picker, so perhaps you simply need to ask the user to choose a storage location for your particular data files.

How exactly to declare file type associations is a form of contracts, so we'll cover the details in Chapter 12.

Rich Enumeration with File Queries

When you are ready to enumerate files within a particular location, what you'll be using for that purpose is a file query or, simply, a search. This is essential if you're going to show files in a UI of some kind, because file queries account for the fact that you normally want more metadata for the files you enumerate than just the file or pathnames along. Windows 8 in general is very visual, and so file queries strongly support acquiring visual data along with the textual or factual.

Queries always start with a `StorageFolder`, whose `createFileQuery[WithOptions]`, `createFolderQuery[WithOptions]`, and `createItemQuery[WithOptions]` methods (6 total) provide for enumerating files, folders, or both, within whatever folder the `StorageFolder` is attached to.

That's at least the simple way of looking at it! On the deepest level, these APIs they are capable of working with [Advanced Query Syntax \(AQS\)](#) searches on folder contents, going deep into all the subfolders as well.⁴⁴ They tie into a set of search features available in the [Windows.Storage.Search](#) namespace, which opens up many vistas that we won't be able to fully explore here (the rabbit hole goes very deep!). What's most helpful to understand is that file queries let you retrieve collections of files in many different "shapes" such as a flat list, a hierarchy, and various sort orders including those oriented around media properties. In addition, file queries also provide for obtaining thumbnails as well as automatic retrieval of album art for music.

We'll be looking at some of the media-specific feature of file queries in Chapter 10. Here, let's concentrate on understanding how file queries work, starting with the basics that are demonstrated in the `FileQuery` example included with this chapter's companion content. This example is a copy of the [Query sample](#) from the Windows SDK, which itself has only one scenario oriented on the music library that lets you enter in an AQS string directly. However, this isn't always what you'll be using in an app, so I wanted to show many other variations.

The simplest queries are created with the base `StorageFolder.create*` methods and no parameters:

```
folder.createFileQuery();
folder.createFolderQuery();
folder.createItemQuery();
```

⁴⁴ Note that contrary to any examples in the docs, apps should *always* use the full name of [Windows properties](#) in queries such as `System.ItemDate`: rather than the user-friendly shorthand such as `date`: because the latter will not work on localized builds of Windows.

The first two methods here are actually just shortcuts for the one-parameter variants with the same names, where that parameter is a value from the `Windows.Storage.Search.CommonFileQuery` or `CommonFolderQuery` enumerations. These shortcut versions just use the `defaultQuery` value for a simple alphabetical, shallow enumeration of the folder contents. `createItemQuery`, for its part, has only this one form.

Creating a query itself doesn't actually enumerate anything until you ask it to through an async method: for file queries, the method is `getFilesAsync`; for folders it's `getFoldersAsync`; and for items it's `getItemsAsync`. (See a pattern here?) So in Scenario 2 of the FileQuery example I have these three functions attached to buttons:

```
function fileQuery() {
    var query = picturesLibrary.createFileQuery();
    SdkSample.showResults(query.getFilesAsync());
}

function folderQuery() {
    var query = picturesLibrary.createFolderQuery();
    SdkSample.showResults(query.getFoldersAsync());
}

function itemQuery() {
    var query = picturesLibrary.createItemQuery();
    SdkSample.showResults(query.getItemsAsync());
}
```

where the `SdkSample.showResults` function in `default.js` just takes the promise from each async operation, calls its `done` method, and creates a listing of the items in the collection. Running this sample you'll see then a list of files and/or folders in your pictures library in the app's output area.

Tip The actual object types returned by these `create*Query` APIs are `StorageFileQueryResult`, `StorageFolderQueryResult`, and `StorageItemQueryResult`, all in the `Windows.Storage.Search` namespace. These all provide some additional properties like `folder`, methods like `findStartIndexAsync` and `getItemCountAsync`, and events like `optionschanged` and `contentschanged`. The latter event especially is something you can use to monitor changes to the file system that affect query results.

Beyond this shallow default behavior, file and folder queries have many other possibilities as expressed in the `CommonFileQuery` and `CommonFolderQuery` enumerations:

- `CommonFileQuery`: `orderByname`, `orderByTitle`, `orderByDate`, `orderByMusicProperties`, and `orderBySearchRank`.
- `CommonFolderQuery`: `groupByType`, `groupByTag`, `groupByAuthor`, `groupByYear`, `groupByMonth`, `groupByArtist`, `groupByComposer`, `groupByGenre`, `groupByPublishedYear`, and `groupByRating`.

Clearly, the effect of these choices depends on the queried items actually containing metadata that

would support the ordering or grouping, but it is allowable to query all folders for all types of files and folders. To demonstrate this, Scenario 3 of the FileQuery example lets you choose the music, pictures, or videos library; whether to query files or folders; and an applicable common query to apply; and then run a search to see the results (for whatever files you have in your music, pictures, and videos libraries, of course). Do note that using `orderBySearchRank` with files isn't really meaningful in this context because it's meant to work with AQS-based searches. We'll see this a little later. (Also—call me a slacker!—the results of a grouped folder query isn't very interesting when one doesn't group the display output, but for an example of that you can refer to Scenario 2 of the [Enumerate files and folders in a location sample](#).)

The code in `scenario3.js` is pretty much just the mechanics of mapping your UI selections to either `createFileQuery` or `createFolderQuery` with the right parameters, so there's no need to look at most of it here. The one piece that is important to point out is the use of the `isCommonFileQuerySupported` and `isCommonFolderQuerySupported` methods of `StorageFolder`. These are used to test whether the current folder will actually support the particular query you want to try:

```
if (folder.isCommonFileQuerySupported(selectedQuery)) {
    query = folder.createFileQuery(selectedQuery);
    if (query) {
        promise = query.GetFilesAsync();
    }
}
```

You'll find when running the sample that in the media libraries, at least, all the common file and folder queries are supported, but that might not be true for all `StorageFolder` objects you might encounter. Remember, for example, that the folder picker might give you a `StorageFolder` from a provider whose data is off in some online service or a database, in which case certain queries might not work.

A similar method, `StorageFolder.areQueryOptionsSupported`, also exists to tests support for custom queries beyond the common ones. A custom query is described by a [Windows.Storage.Search.QueryOptions](#) object (the common queries are just prepopulated instances of these) and is created by passing such an object to the `createFileQueryWithOptions`, `createFolderQueryWithOptions`, and `createItemQueryWithOptions`.

A `QueryOptions` is generally created from scratch using the new operator, after which you populate its properties. You can also use `new QueryOptions(<CommonFolderQuery>)` to retrieve the object for one of the common folder queries, and `new QueryOptions(<CommonFileQuery> [, <file type filter>])` to do the same for common file queries. In this latter case, an optional array of file types can also be given; this is a shortcut to quickly customize a common query with a specific set of file types. Without it, the filter is set to `"*"` by default. That is, if you wanted to just find `.mp3` files in your music library ordered by title, you would use this kind of code (see `scenario4.js` in the FileQuery example):

```
var musicLibrary = Windows.Storage.KnownFolders.musicLibrary;
var options = new Windows.Storage.Search.QueryOptions(
    Windows.Storage.Search.CommonFileQuery.orderByTitle, [".mp3"]);
```



```

if (musicLibrary.areQueryOptionsSupported(options)) {
    var query = musicLibrary.createFileQueryWithOptions(options);
    SdkSample.showResults(query.GetFilesAsync());
}

```

If you create a `QueryOptions` from scratch, you can set a number of options. The more general or basic ones are as follows:⁴⁵

- `fileTypeFilter` An vector of strings that describe the desired file type extensions, as in ".mp3". The default is an empty list (no filtering).
- `folderDepth` Either `Windows.Storage.Search.FolderDepth.shallow` (the default) or `deep`.
- `indexerOption` A value from `Windows.Storage.Search.IndexerOption`, which is one of `useIndexerWhenAvailable`, `onlyUseIndexer` (limit the search to indexed content only), and `doNotUseIndexer` (query the file system directly bypassing the indexer). As the latter is the default, you'll typically want to explicitly set this property to `useIndexerWhenAvailable`.
- `sortOrder` A vector of `Windows.Storage.Search.SortEntry` structures that each contain a Boolean named `ascendingOrder` (false for descending order) and a `propertyName` string. Each entry in the vector defines a sort criterion; these are applied in the order they appear in the vector. An example of this will be given a little later.

Three of the `QueryOptions` properties then apply to searches with AQS strings:

- `applicationSearchFilter` An AQS string.
- `userSearchFilter` Another AQS string.
- `language` A string containing the BCP-47 language tag associated with the AQS strings.

When the query is built through a method like `createFileQueryWithOptions`, the application and user filter strings here are combined. What this means is that you can separately manage any filter you want to apply generally for your app (`applicationSearchFilter`) from user-supplied search terms (`userSearchFilter`). This way you can enforce some search filters without requiring the user to type them in, and without always having to combine strings yourself.

As noted before, the `CommonFileQuery.orderBySearchRank` query is meaningful only when combined with an AQS string, which is to say that keyword-based searches return ranked results for which this common file query would apply. Returning to Scenario 1 of the Query sample, then, we see how it uses this ordering along with the `userSearchFilter` property:

⁴⁵ Another property, `dateStackOption` (a value from `Windows.Storage.Search.DateStackOption`), is read-only within this structure but can be set when creating a `QueryOptions` from a `CommonFolderQuery`.

```

var musicLibrary = Windows.Storage.KnownFolders.musicLibrary;
var options = new Windows.Storage.Search.QueryOptions(
    Windows.Storage.Search.CommonFileQuery.orderBySearchRank, ["*"]);
options.userSearchFilter = searchFilter;
var fileQuery = musicLibrary.createFileQueryWithOptions(options);

```

On my machine, where I have a number of songs with “Nightingale” in the title, as well as an album called “Nightingale Lullaby,” a search using the string *"Nightingale" System.ItemType: "mp3"* in the above code gives me results that look like this in the sample:⁴⁶

24 files found

```

04 Song Of The Nightingale2.mp3
16 Song Of The Nightingale (Instrumental).mp3
01 Song of the Nightingale.mp3
04 Song of the Nightingale.mp3
12 Song Of The Nightingale.mp3
- 18 - Twinkle Twinkle.mp3
- 17 - Tum-Balayka.mp3
- 16 - Tina and Gina Bobina.mp3
- 15 - Shayna's Lullaby.mp3
- 14 - Rockabye Baby.mp3
- 13 - Lullaby for Carol.mp3

```

This shows that the search ranking favors songs with “Nightingale” directly in the title, but also include those from an album with that name.

My search string here, by the way, shows how you might use the [applicationSearchFilter](#) and [userSearchFilter](#) properties together. If my app was capable of working only with mp3 or some other formats, I could store *"type: 'mp3'"* in [applicationSearchFilter](#) and store user-provided terms like *"Nightingale"* in [userSearchFilter](#). This way I avoid having to join them manually in my code.

Beyond the properties that you set within a [QueryOptions](#) object, it also has some information and capabilities of its own. The [groupPropertyName](#), for one, is a string property that indicates the type of property that the query is being grouping by. You can also retrieve the query options as a string using the [saveToString](#) method and recreate the object from a string using [loadFromString](#) (that is, the analog of [JSON.stringify](#) and [JSON.parse](#)).

The [setPropertyPrefetch](#) method goes even deeper still, allowing you to indicate a group of file properties that you want to optimize for fast retrieval—they’re accessed through the same APIs as file properties in general, but they come back faster, meaning that if you’re displaying a collection of files in a [ListView](#) using a custom data source with certain properties from enumerated files, you’d want to set those up for prefetch so that the control renders faster. (The [WinJS.UI.StorageDataSource](#) does this already.) Similarly, [setThumbnailPrefetch](#) tells Windows what kinds of thumbnails you want to include in the enumeration—again, you can ask for these without setting the prefetch, but they come

⁴⁶ For more on AQS, again refer to the [Advanced Query Syntax \(AQS\)](#) topic as well as [Using Advanced Query Syntax Programmatically](#). Again, then creating queries programmatically, be sure to use the full name like *System.ItemType* and not the shorthand form like *type* to identify the desired [Windows properties](#).

back faster when you do. This again helps you optimize the display of a file collection.⁴⁷

We briefly saw similar usage of thumbnail properties back in Chapter 5, when we took advantage of a shortcut to the pictures library with `WinJS.UI.StorageDataSource` and could specify a thumbnail size option:

```
myFlipView.itemDataSource = new WinJS.UI.StorageDataSource("Pictures",
    { requestedThumbnailSize: 480 });
```

A more general example that also includes the `QueryOptions.sortOrder` vector can be found in the [StorageDataSource and GetVirtualizedFilesVector sample](#), which got a footnote in Chapter 5. In its `scenario2.js` we see the creation of a `QueryOptions` from scratch, setting up two `sortOrder` criteria, and setting up thumbnail options in the data source:

```
function loadListViewControl() {
    // Build datasource from the pictures library
    var library = Windows.Storage.KnownFolders.picturesLibrary;
    var queryOptions = new Windows.Storage.Search.QueryOptions;
    // Shallow query to get the file hierarchy
    queryOptions.folderDepth = Windows.Storage.Search.FolderDepth.shallow;
    queryOptions.sortOrder.clear();
    // Order items by type so folders come first
    queryOptions.sortOrder.append({ ascendingOrder: false, propertyName: "System.IsFolder" });
    queryOptions.sortOrder.append({ ascendingOrder: true, propertyName: "System.ItemName" });
    queryOptions.indexerOption = Windows.Storage.Search.IndexerOption.useIndexerWhenAvailable;

    var fileQuery = library.createItemQueryWithOptions(queryOptions);
    var dataSourceOptions = {
        mode: Windows.Storage.FileProperties.ThumbnailMode.picturesView,
        requestedThumbnailSize: 190,
        thumbnailOptions: Windows.Storage.FileProperties.ThumbnailOptions.none
    };

    var dataSource = new WinJS.UI.StorageDataSource(fileQuery, dataSourceOptions);

    // Create the ListView...
};
```

If you're really interested in digging deeper here, you can look at how `StorageDataSource` sets up file queries; just search for this class in the `ui.js` file of `WinJS` and you'll find it. Along the way, you'll run into one more set of WinRT APIs—perhaps the bottom of the hole!—that I wanted to mention before wrapping up this subject: `Windows.Storage.BulkAccess`. These actually exist solely for use by `StorageDataSource` and are not intended for direct use in apps. Even if you create your own data source or collection control, it's best to just use the enumeration and prefetch APIs we've already discussed, as they give identical performance.

⁴⁷ See [What's Changed for App Developers Since the Consumer Preview](#) on the Windows 8 Developer Blog for a few more details on these.

Here My Am! Update

To bring together some of the topics we've covered in this chapter, the companion content includes another revision of the Here My Am! app with the following changes and additions (mostly to `home.js` unless notes):

- It now incorporates the [Bing Maps SDK](#) so that the control is part of the package rather than loaded from a remote source. This eliminates the `iframe` we've been using to host the map, so all the code from `html/map.html` can move into `default.js`. Note that to run this sample in Visual Studio you need to download and install the SDK yourself.
- Instead of copying pictures taken with the camera to app data, those are now copied to a HereMyAm folder in the Pictures library. The Pictures Library capability has been added in the manifest.
- An `AppBar` command now allows you to use the File Picker to select an image to load instead of relying solely on the camera. This also allows you to use a camera app, if desired. Note that we use a particular `settingsIdentifier` with the picker in this case to distinguish from the picker for recent images.
- Another `AppBar` command allows you to choose from recent pictures from the camera. This defaults to our folder in the Pictures library and uses a different `settingsIdentifier`.
- Additional commands for About, Help, and a Privacy Statement are included on the Settings pane using the `WinJS.Application.onsettings` event (see `default.js`). The first two display content from within the app whereas the third pulls down web content in an `iframe`; all the settings pages are found in the `html` folder of the project.

What We've Just Learned

- Statefulness is important to WinRT apps, to maintain a sense of continuity between sessions even if the app is suspended and terminated.
- App data is session, local, temporary, and roaming state that is tied to the existence of an app; it is accessible only by that app.
- User data is stored in locations other than app data (such as the user's music, pictures, videos, and documents libraries) and persists independent of any given app, and multiple apps might be able to open and manipulate user files.
- App data is accessed through the `Windows.Storage.ApplicationData` API and accommodates both structured settings containers as well as file-based data. Additional APIs like IndexedDB and HTML5 `localStorage` are also available.

- It is important to version app state, especially where roaming is concerned, as versioning is how the roaming service manages what app state gets roamed to which devices based on what version apps are looking for.
- Roaming state is limited to about 100K, otherwise Windows will not roam the data. Services like SkyDrive can be used to roam larger files, including user data.
- The typical roaming period is 30 minutes or less. A single setting or composite named “HighPriority,” so long as it’s under 8K, will be roamed within a minute.
- The [StorageFolder](#) and [StorageFile](#) classes in WinRT are really the core object for working with folders and files. All programmatic access to the file system begins, in fact, with a [StorageFolder](#). Otherwise, the user can point to files and folders through the file picker API, which is really the first choice for file access.
- Blobs are also very useful aids in working with files, and WinRT provide some simpler APIs in the [Windows.Storage.FileIO](#) and [PathIO](#) classes. WinJS offers some simplified methods for reading and writing text files (especially in conjunction with app state), and the HTML5 [FileReader](#) is supported.
- WinRT offers encryption services through [Windows.Security.Cryptography](#), as well as a built-in compression mechanism in [Windows.Storage.Compression](#).
- To use the Settings pane, an app populates the top-level pane provided by Windows with specific commands. Those commands map to handlers that either open a hyperlink (in a browser) or display a settings flyout using the [WinJS.UI.SettingsFlyout](#) control. Those flyouts can contain any HTML desired, including [iframe](#) elements that load remote content.
- Access to user data folders, such as media libraries, documents, and removable storage, is controlled by manifest capabilities. Such capabilities need be declared only if the app needs to access the file system in some way other than using the file picker.
- The file picker is the way that users can select files from any safe location in the file system, as well as files that are provided by other apps (where those files might be remote, stored in a database, or otherwise not present as file entities on the local file system). The ability to select files directly from other apps—including files that another app might generate on demand—is one of the most convenient and powerful features of WinRT apps.
- [StorageFolder](#) objects provide a very rich and extensive capability to query and search its contents through file queries. These queries can be simple to complex and can employ Advanced Query Syntax (AQS) search strings.

Chapter 9

Input and Sensors

Touch is clearly one of the most exciting means of interacting with a computer that has finally come of age. Sure, we've had touch-sensitive devices for many years: I remember working with a touch-enabled screen in my college days, which I have to admit is almost an embarrassingly long time ago now! In that case, the touch sensor was a series of transparent wires embedded in a plastic sheet over the screen, with an overall touch resolution of around 60 wide by 40 high...and, to really date myself, the monitor itself was only a text terminal!

Fortunately, touch screens have progressed tremendously in recent years. They are responsive enough for general purpose use (that is, you don't have to stab them to register a point), are built into high-resolution displays, are relatively inexpensive, and are capable of doing something more than replicating the mouse—namely, supporting multitouch and sophisticated gestures.

Great touch interaction is thus now a fundamental feature of great apps, and designing for touch in many ways means thinking through UI concerns anew. In your layout, for example, it means making hit targets a size that's suitable for a variety of fingers. In your content navigation, it means utilizing direct gestures such as swipes and pinches rather than relying on only item selection and navigation controls. Similarly, designing for touch means thinking through how gestures might enrich the user experience—and also how to provide for discoverability and user feedback that has generally relied on mouse-only events like hover.

All in all, approach your design as if touch was the *only* means of interaction that your users might have. At the same time, it's very important to remember that new methods of input seldom obsolete existing ones. Sure, punch cards did eventually disappear, but the introduction of the mouse did not obsolete keyboards. The availability of speech recognition or handwriting has obsoleted neither mouse nor keyboard. I think the same is true for touch: it's really a complementary input method that has its own particular virtues but is unlikely to wholly supplant the others. As Bill Buxton of Microsoft Research has said, "Every modality, including touch, is best for something and worst for something else." I expect, in time, we'll see ourselves using keyboard, mouse, and touch together, just as we learned to integrate the mouse in what was once a keyboard-only reality.

Windows is designed to work well with all forms of input—to work great with touch, to work great with mice, to work great with keyboards, and, well, to just work great on diverse hardware! For this reason, Windows provides a unified pointer-based input model wherein you can differentiate the different inputs if you really need to but can otherwise treat them equally. You can also focus more on higher-level gestures as well, which can arise from any input source, and not worry about raw pointer events at all. Indeed, the very fact that we haven't even brought this subject up until now, midway through this book, gives testimony to just how natural it is to work with all kinds of pointer input without having to think about it: the controls and other UI elements we've been using have done all

that work for us. Handling such events ourselves thus arises primarily when creating your own controls or otherwise doing direct manipulation of noncontrol objects.

The keyboard also remains an important consideration, and this means both hardware keyboards and the on-screen “soft” keyboard. The latter has gotten more attention in recent years for touch-only devices but actually has been around for some time for accessibility purposes. In Windows, too, the soft keyboard includes a handwriting recognizer—something apps just get for free. And when an app wants to work more closely with raw handwriting input—known as ink—those capabilities are present as well.

The other topic we’ll cover in this chapter is sensors. It might seem an incongruous subject to place alongside input until you come to see that sensors, like touch screens themselves, *are* another form of input! Sensors tell an app what’s happening to the device in its relationship to the physical world: how its positioned in space (relative to a number of reference points), how it’s moving, how it’s being held relative to its “normal” orientation, and even how much light is shining on it. Thinking of sensors in this light (pun intended), we begin to see opportunities for apps to directly integrate with the world around a device rather than requiring users to tell the app about those relationships in some more abstract way. And just to warn you, once you see just how easy it is to use the WinRT APIs for sensors, you might be shopping for a new piece of well-equipped hardware!

Touch, Mouse, and Stylus Input

Where pointer-based input is concerned—which includes touch, mouse, and pen/stylus input—the singular message from Microsoft has been (and remains), “Design for touch and get mouse and stylus for free.” This is very much the case, as we shall see, but we’ve also found that a phrase like “touch-first design” that sounds great to a consumer can be a terrifying proposition for developers! With all the attention around touch, consumer expectations are often very demanding, and meeting such expectations seems like it will take a lot of work.

Fortunately, Windows 8 provides a unified framework for handling pointer input—from all sources—such that you don’t actually need to think about the differences until you truly need to. In this way, touch-first design really *is* a design issue more than an implementation issue.

We’ll talk more about designing for touch in the next section. What I wanted to discuss first is how you as a developer should approach implementing those designs once you have them so that you don’t make any distinctions between the types of pointer input until it’s necessary:

- First, *use templates and standard controls* and you get lots of touch support for free, along with mouse, pen, stylus, and keyboard support. If you build up your UI with standard controls, set appropriate attributes for tab order (for keyboard users), and handle standard DOM events like [click](#), you’re pretty much covered. Controls like semantic zoom already handle different kinds of input (as we saw in Chapter 5, “Collections and Collection Controls”), and other CSS styles like snap points and content

zooming automatically handle various interaction gestures.

- Second, when you need to handle gestures directly, as with custom controls or other elements with which the user will interact directly, *use the gesture events* like [MSGestureTap](#) and [MSGestureHold](#) along with event sequences for inertial gestures ([MSGestureStart](#), [MSGestureChange](#), [MSGestureEnd](#)). The benefit here is that gestures are essentially higher-order interpretations of a series of lower-level pointer events, meaning that you don't have to do such interpretation yourself. For example, a pointer down followed by a pointer up within a certain movement threshold (to account for wiggling fingers) becomes a single tap gesture. A pointer down followed by a short drag followed by a pointer up becomes a swipe that triggers a series of events, possibly including inertial events (ones that continue to fire even after the pointer, like a touch point, is physically released). Note that if you want to capture and save pointer input directly without concern for gestures, there is also built-in support for *inking*, as we'll see later on.
- Third, if you need to handle pointer events directly, *use the unified pointer events* like [MSPointerDown](#), [MSPointerMove](#), and so forth. These are lower-level events than gestures, and they are primarily appropriate for apps that don't necessarily need gesture interpretation. For example, a drawing app simply needs to trace different pointers with on-screen feedback, and concepts like swipe and inertial gestures aren't meaningful. Pointer events also provide more specialized device data such as pressure, rotation, and tilt, which is surfaced through the pointer events. Still, it is possible to implement gestures directly with pointer events, as a number of the built-in controls do.
- Finally, an app can *work directly with the gesture recognizer* to provide its own interpretations of pointer events into gestures.



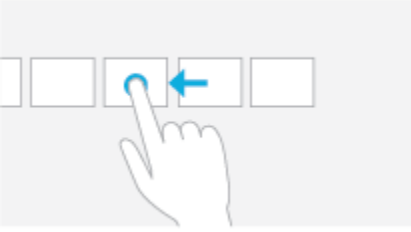
So, what about legacy DOM events that we already know and love, beyond [click](#)? Can you still work with the likes of [mousedown](#), [mouseup](#), [mouseover](#), [mousemove](#), [mouseout](#), and [mousewheel](#)? The answer is yes, because pointer events from all input sources will be automatically translated into these legacy events. This can be useful when you're porting code from a web app into a WinRT 8 app, for example. This translation takes a little extra processing time, however, so for new code you'll generally realize better responsiveness by using the gesture and pointer events directly. Legacy mouse events also assume a single pointer, so you won't be able to distinguish multiple simultaneous touch points—it'll just look like the user is very rapidly clicking around the screen! As much as possible, use the gesture and pointer events in your code.

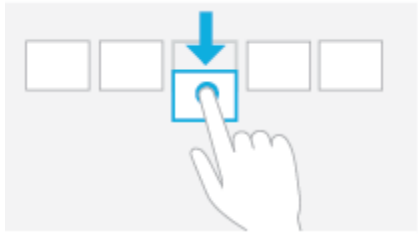
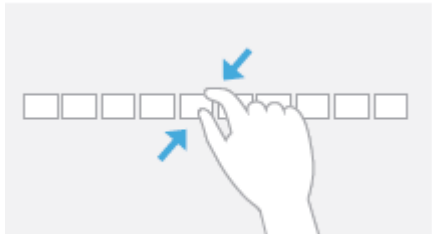


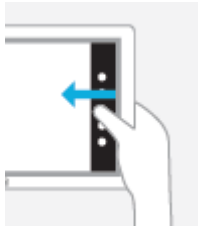
The Touch Language, Its Translations, and Mouse/Keyboard Equivalents

On the Windows Developer Center, the rather extensive article on [Touch interaction design](#) is helpful

for designers and developers alike. It discusses various ergonomic considerations, has some great diagrams on the sizes of human fingers, provides clear guidance on the proper size for touch targets given that human reality, and outlines key design principles such as providing direct feedback for touch interaction (animation) and having content follow your finger.

Most importantly, the design guidance also describes the Windows 8 Touch Language, which contains the eight core gestures that are baked into the system and the controls. The table below shows and describes the gestures and indicates what events appear in the app for them.

Gesture	Meaning and Gesture Events	Description
<p>One finger touches the screen and lifts up.</p> 	<p>Tap for primary action (commanding); appears as <code>click</code> and <code>MSGestureTap</code> events on the element.</p>	<p>Tapping on an element invokes its primary action, typically executing a command, checking a box, setting a rating, positioning a cursor, etc.</p>
<p>One finger touches the screen and stays in place.</p> 	<p>Press and hold to learn; appears as <code>contextmenu</code> and <code>MSGestureHold</code> events on the element.</p>	<p>This touch interaction causes detailed information or teaching visuals (for example, a tooltip or context menu) to be displayed without a commitment to an action. Anything displayed this way should not prevent users from panning if they begin sliding their finger.</p>
<p>One or more fingers touch the screen and move in the same direction.</p> 	<p>Slide to pan (can be horizontal or vertical); appears as scrolling events as well as a gesture series (<code>MSGestureStart</code>, <code>MSGestureChange</code>, <code>MSGestureEnd</code>, possibly with inertial gesture events, as well as <code>MSPointer*</code> events).</p>	<p>Slide is used primarily for panning interactions but can also be used for moving, drawing, or writing. Slide can also be used to target small, densely packed elements by scrubbing (sliding the finger over related objects such as radio buttons).</p>

<p>One or more fingers touch the screen and move a short distance in the same direction.</p> 	<p>Swipe to select, command, and move (can be horizontal or vertical)—also called <i>cross-slide</i>; appears as a gesture series (MSGestureStart, MSGestureChange, MSGestureEnd, as well as MSPointer* events).</p>	<p>Sliding the finger a short distance, perpendicular to the panning direction, selects objects in a list or grid; also implies displaying commands in an app bar relevant to the selection.</p>
<p>Two or more fingers touch the screen and move closer together or farther apart.</p> 	<p>Pinch and stretch to zoom; appears as a gesture series (MSGestureStart, MSGestureChange, MSGestureEnd), but apps can use the <code>-ms-content-zooming: zoom</code> and <code>-ms-touch-action: pinch-zoom</code> CSS styles to enable touch zooming automatically.</p>	<p>Can be used for optical zoom or resizing, as well as for semantic zoom where applicable.</p>
<p>Two or more fingers touch the screen and move in a clockwise or counter-clockwise arc.</p> 	<p>Turn to rotate; appears as a gesture series (MSGestureStart, MSGestureChange, MSGestureEnd).</p>	<p>Rotates an object or a view.</p>
	<p>Swipe from top or bottom edge for app commands; handled automatically through the AppBar control, though an app can also detect these events directly through Windows.UI.Input.EdgeGesture.</p>	<p>The bottom app bar contains app commands for the current page context; the top app bar provides for navigation, if applicable.</p>
	<p>Swipe from edge for system commands; handled automatically by the system with the app receiving specific events related to the selected charm, when applicable.</p>	<p>Swiping from the right displays the Charms bar; swiping from the left cycles through currently running apps; swiping from the top edge to the bottom closes the current app; swiping from the top edge to the left or right snaps the current app to one side of the screen.</p>

Additional details and guidelines for designing around this touch language can be found on the [Gestures, manipulations, and interactions](#) topic.

You might notice in the table above that many of the gestures in the touch language don't actually have a single event associated with them (like pinch or rotate) but are instead represented by a series of gesture or pointer events. The reason for this is that these gestures, when used with touch, typically involve animation of the affected content while the gesture is happening. Swipes, for example, show linear movement of the object being panned or selected. A pinch or stretch movement will often be actively zooming the content. (Semantic Zoom is an exception, but then you just let the control handle the details.) And a rotate gesture should definitely give visual feedback. In short, handling these gestures with touch, in particular, means dealing with a series of events rather than just a single one, as we shall see.

This is one reason that it's so helpful (and time-saving!) to use the built-in controls as much as possible, because they already handle all the gesture details for you. The ListView control, for example, contains all the pointer/gesture logic to handling pans and swipes, along with taps. The Semantic Zoom control, like I said, implements pinch and stretch by watching `MSPointer*` events. If you look at the source code for these controls within WinJS, you'll start to appreciate just how much they do for you (and what it will look like to implement a rich custom control of your own, using the gesture recognizer!).

On the theme of "write for touch and get other input for free" all of these gestures also have mouse and keyboard equivalents, which the built-in controls also provide for you. It's also helpful to know what those equivalents are, as shown in the table below. The "Standard Keystrokes" section later in this chapter also lists many other command-related keystrokes.

Touch	Keyboard	Mouse	Pen/Stylus
Press and hold (or tap on text selection)	Right-click button	Right button click	Press and hold
Tap	Enter	Left button click	Tap
Slide (short distance)	Arrow keys	Left button click and drag, click on scrollbar arrows, drag the scrollbar thumb, use the mouse wheel	Tap on scrollbar arrows, drag scrollbar thumb, tap and drag
Slide + inertia (long distance)	Page Up/Page Down	Left button click and drag, click on scrollbar track, drag the scrollbar thumb, use the mouse wheel	Tap on scrollbar track, drag scrollbar thumb, tap and drag
Swipe to select	Right-click button or spacebar	Right button click	Tap and drag
Pinch/Stretch	Ctrl+ and Ctrl-	Ctrl+mouse wheel or UI command	UI command or other hardware feature
Swipe from edge	Win+Z, Win+Tab, Win+C or Win+Shift+C	Clicking on corners of the screen; right-click shows app bar	Drag in from edge
Rotate	Ctrl+, and Ctrl+.	Ctrl+Shift+mouse wheel	UI command or other hardware feature

You might notice a conspicuous absence of double-click and/or double-tap gestures in this list. Does that surprise you? In early builds of Windows 8 we actually did have a double-tap gesture, but it

turned out to not be all that useful and sometimes very difficult for users to perform. I can say from watching friends over the years that double-clicking with the mouse isn't even all it's cracked up to be. People with not-entirely-stable hands will often move the mouse quite a ways between clicks, just as they might move their finger between taps. As a result, the reliability of a double-tap ends up being pretty low, and since it wasn't really needed in the touch language, it was simply dropped altogether.

Sidebar: Creating Completely New Gestures?

While the Windows 8 touch language provides a simple yet fairly comprehensive set of gestures, it's not too hard to imagine other possibilities. The question is, when is it appropriate to introduce a new kind of gesture or manipulation?

First, it makes sense that apps don't generally introduce new ways to do the same things, such as additional gestures that just swipe, zoom, etc. It's better to simply get more creative in how the app interprets an existing gesture. For example, a swipe gesture might pan a scrollable region but can also just move an object on the screen—no need to invent a new gesture.

Second, if you have controls placed on the screen where you want the user to give input, there's no need to think in terms of gestures at all: just apply the input from those controls appropriately.

Third, even when you do think a custom gesture is needed, the bottom-line recommendation is to make those interactions feel natural, rather than something you just invent for the sake of invention. We also recommend that gestures behave consistently with the number of pointers, velocity/time, and so on. For example, separating an element into three pieces with a three-finger stretch and into two pieces with a two-finger stretch is fine; having a three-finger stretch enlarge an element while a two-finger stretch zooms the canvas is a bad idea, because it's not very discoverable. Similarly, the speed of a horizontal or vertical flick can affect the velocity of an element's movement, but having a fast flick switch to another page while a slow flick highlights text is a bad idea. In this case, having different functions based on speed creates a difficult UI for your customers because they'll all have different ideas about what "fast" and "slow" mean and might be limited by their physical abilities.

Finally, with any custom gesture, recognize that you are potentially introducing an inconsistency between apps. When a user starts interacting with a certain kind of app in a new way, he or she might start to expect that of other apps and might become confused (or upset) when those apps don't behave in the same way, especially if those apps use a similar gesture for a completely different purpose! Complex gestures, too, as hinted above, might be difficult for some, if not many, people to perform; might be limited by the kind of hardware in the device (number of touch points, responsiveness, etc.); and are generally not very discoverable. In most cases it's probably far simpler to just add an appbar command on a button on your app canvas to accomplish the same goal.

Edge Gestures

As we saw in Chapter 7, “Commanding UI,” you don’t need to do anything special for app commands on the app bar or navigation bar to appear: Windows automatically handles the edge swipe from the top and bottom of your app, along with right-click, Win+Z, and the context menu key on the keyboard. That said, you can detect when these events happen directly by listening for the `starting`, `completed`, and `canceled` events on the [Windows.UI.Input.EdgeGesture](#) object:

```
var edgeGesture = Windows.UI.Input.EdgeGesture.getForCurrentView();
edgeGesture.addEventListener("starting", onStart);
edgeGesture.addEventListener("completed", onComplete);
edgeGesture.addEventListener("canceled", onCancel);
```

With these, `completed` fires for all input types; the `starting` and `canceled` events occur only for touch. Within these events, the `eventArgs.kind` property contains a value from the `EdgeGestureKind` enumeration that indicates the kind of input that invoked the event. The `starting` and `canceled` events will always have the kind of touch, obviously, whereas `completed` can be any `touch`, `keyboard`, or `mouse`:

```
function onComplete(e) {
    // Determine whether it was touch or keyboard invocation
    if (e.kind === Windows.UI.Input.EdgeGestureKind.touch) {
        id("ScenarioOutput").innerText = "Invoked with touch.";
    }
    else if (e.kind === Windows.UI.Input.EdgeGestureKind.mouse) {
        id("ScenarioOutput").innerText = "Invoked with right-click.";
    }
    else if (e.kind === Windows.UI.Input.EdgeGestureKind.keyboard) {
        id("ScenarioOutput").innerText = "Invoked with keyboard.";
    }
}
```

The code above is taken from Scenario 1 of the [Edge gesture invocation sample](#). In Scenario 2, the sample also shows that you can prevent the edge gesture event from occurring for a particular element if you handle the `contextmenu` event for that element and call `eventArgs.preventDefault` in your handler. It does this for one element on the screen, such that right-clicking that element with the mouse or pressing the context menu key when that element has the focus will prevent the edge gesture events:

```
document.getElementById("handleContextMenuDiv").addEventListener("contextmenu", onContextMenu);

function onContextMenu(e) {
    e.preventDefault();
    id("ScenarioOutput").innerText =
        "The ContextMenu event was handled. The EdgeGesture event will not fire.";
}
```

Note that this method has no effect on edge gestures via touch and does not affect the Win+Z key combination that normally invokes the app bar. It’s primarily to show that if you need to handle the `contextmenu` event specifically, you usually want to prevent the edge gesture.

CSS Styles That Affect Input

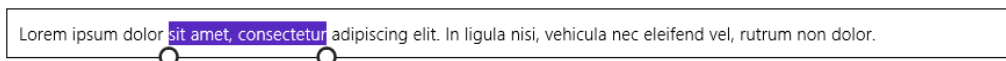
While we're on the subject of input, it's a good time to mention a number of CSS styles that affect the input an app might receive.

One style is `-ms-user-select`, which we've encountered a few times already in Chapter 3, "App Anatomy and Page Navigation," and Chapter 4, "Controls, Control Styling, and Data Binding." This style can be set to one of the following:

- `none` disables direct selection, though the element as a whole can be selected if it's parent is selectable.
- `inherit` sets the selection behavior of an element to match its parent.
- `text` will enable selection for text even if the parent is set to `none`.
- `element` enables selection for an arbitrary element.
- `auto` (the default) may or may not enable selection depending on the control type and the styling of the parent. For an element that is not a text control and does not have `contenteditable="true"`, it won't be selectable unless it's contained within a selectable parent.

If you want to play around with the variations, refer to the [Unselectable content areas with -ms-user-select CSS attribute sample](#), which wins the prize for the longest sample name in the entire Windows SDK!

A related style, but one not shown in the sample, is `-ms-touch-select`, which can be either `none` or `grippers`, the latter being the style that enables the selection control circles for touch:



Selectable text elements automatically get this style, as do other textual elements with `contenteditable="true"`—you can thus use `-ms-touch-select` to turn them off. To see the effect, try this with some of the elements in scenario 1 of the aforementioned sample with the really long name!

In Chapter 6, "Layout," we introduced the idea of snap points for panning, with the `-ms-scroll-snap*` styles. Along these same lines, listed on the [Touch: Zooming and Panning](#) styles reference, are others for content zooming, such as `-ms-content-zooming` and the `-ms-content-zoom*` styles that provide snap points for zoom operations as well. The important thing is that `-ms-content-zooming: zoom` (as opposed to the default, `none`) enables automatic zooming with touch and the mouse wheel, provided that the element in question allows for overflow in both x and y dimensions. There are quite a number of variations here for panning and zooming, and how those gestures interact with WinJS controls. I'll leave it to the [Input: Pan/scroll and zoom sample](#) to explain the details.

Finally, the `-ms-touch-action` style provides for a number of options on an element:⁴⁸

- `none` Disables touch on the element.
- `auto` Enables usual touch behaviors.
- `pan-x/pan-y` The element permits horizontal/vertical touch panning, which is performed on the nearest ancestor that is horizontally/vertically scrollable, such as a parent `div`.
- `pinch-zoom` Enables pinch-zoom on the element, performed on the nearest ancestor that has `-ms-content-zooming: zoom` and overflow capability. For example, an `img` element by itself won't respond to the gesture with this style, but if you place it in a parent `div` with `overflow` set, it will.
- `manipulation` Shorthand equivalent of `pan-x pan-y pinch-zoom`.

For an example of panning and zooming, try creating a simple app with markup like this (use whatever image you'd like):

```
<div id="imageContainer">
  
</div>
```

and style the container as follows:

```
#imageContainer {
  overflow: auto;
  -ms-content-zooming: zoom;
  -ms-touch-action: manipulation;
}
```

What Input Capabilities Are Present?

The WinRT API in the `Windows.Devices.Input` namespace provides all the information you need about the input capabilities that are available on the current device, specifically through these three objects:

- [MouseCapabilities](#) Properties are `mousePresent` (0 or 1), `horizontalWheelPresent` (0 or 1), `verticalWheelPresent` (0 or 1), `numberOfButtons` (a number), and `swapButtons` (0 or 1).
- [KeyboardCapabilities](#) Contains only a single property: `keyboardPresent` (0 or 1). Note that this does not indicate the presence of the on-screen keyboard, which is always available; `keyboardPresent` specifically indicates a physical keyboard device.
- [TouchCapabilities](#) Properties are `touchPresent` (0 or 1) and `contacts` (a number).

⁴⁸ `double-tap-zoom` is not supported for WinRT apps.

To check whether touch is available, then, you can use a bit of code like this:

```
var tc = new Windows.Devices.Input.TouchCapabilities();
var touchPoints = 0;

if (tc.touchPresent) {
    touchPoints = tc.contacts;
}
```

You'll notice that the capabilities above don't say anything about a stylus or pen. For these and for more extensive information about all pointer devices, including touch and mouse, we have the [Windows.Devices.Input.PointerDevice.getPointerDevices](#) method. This returns an array of [PointerDevice](#) objects, each of which has these properties:

- [pointerDeviceType](#) A value from [Windows.Devices.Input.PointerDeviceType](#) that can be [touch](#), [pen](#), or [mouse](#).
- [maxContacts](#) The maximum number of contact points that the device can support—typically 1 for mouse and stylus and any other number for touch.
- [isIntegrated](#) [true](#) indicates that the device is built into the machine so that its presence can be depended upon; [false](#) indicates a peripheral that the user could disconnect.
- [physicalDeviceRect](#) This [Windows.Foundation.Rect](#) object provides the bounding rectangle as the device sees itself. Oftentimes, a touch screen's input resolution won't actually match the screen pixels, meaning that the input device isn't capable of hitting one and only one pixel. On one of my touch-capable laptops, for example, this resolution is reported as 968x548 for a 1366x768 pixel screen (as reported in [screenRect](#) below). A mouse, on the other hand, typically does match screen pixels one-for-one. Where this could be important for a drawing app that works with a stylus, where an input resolution smaller than the screen would mean there will be some inaccuracy when translating input coordinates to screen pixels.
- [screenRect](#) This [Windows.Foundation.Rect](#) object provides the bounding rectangle for the device on the screen, which is to say, the minimum and maximum coordinates that you should encounter with events from the device. This rectangle will take multimonitor systems into account, and it's adjusted for resolution scaling.
- [supportedUsages](#) An array of [Windows.Devices.Input.PointerDeviceUsage](#) structures that supply what's called HID (human interface device) usage information. This subject is far beyond the scope of this book, so I'll refer you to the [HID Usages](#) page on MSDN for starters.

The [Input Device capabilities sample](#) in the Windows SDK retrieves this information and displays it to the screen through the code in `pointer.js`. I won't show that code here because it's just a matter of iterating through the array and building a big HTML string to dump into the DOM. In the simulator,

the output appears as follows—notice that the simulator reports the presence of touch and mouse both in this case.

```
Output
(1) Pointer Device Type Touch
(1) Is Integrated true
(1) Max Contacts 5
(1) Physical Device Rect 0,0,491.3385925292969,907.0866088867187
(1) Screen Rect 0,0,1366,768
(2) Pointer Device Type Mouse
(2) Is Integrated false
(2) Max Contacts 1
(2) Physical Device Rect 0,0,1366,768
(2) Screen Rect 0,0,1366,768
```

Curious Forge? Interestingly, I ran this same sample in Visual Studio’s Local Machine debugger on a laptop that is definitely not touch-enabled, and yet a touch device was still reported as in the image above! Why was that? It’s because I still had the Visual Studio simulator running, which adds a virtual touch device to the hardware profile. After closing the simulator completely (not just minimizing it), I got an accurate report for my laptop’s capabilities. So be mindful of this if you’re writing code to test for specific capabilities.

Tried remote debugging yet? Speaking of debugging, as mentioned in a sidebar in Chapter 6, “Layout,” testing an app against different device capabilities is a great opportunity to use remote debugging in Visual Studio. If you haven’t done so already, it takes only a few minutes to set up and makes it far easier to test apps on multiple machines. For details, again see on [Running Windows 8 apps on a remote machine](#).

Unified Pointer Events

For any situation where you want to directly work with touch, mouse, and stylus input, perhaps to implement parts of the touch language in this way, you use the `MSPointer*` events. Most art/drawing apps, for example, will use these events to track and respond to screen interaction. Remember again that pointers are a lower-level way of looking at input than gestures, which we’ll see in the next section. Which input model you use depends on the kind of events you’re really looking to work with.

Tip Pointer events won’t fire if the system is trying to do a manipulation like panning or zooming. To disable manipulations on an element, set the `-ms-content-zooming: none` and avoid using `-ms-touch-action` styles of `pan-x`, `pan-y`, `pinch-zoom`, and `manipulation`.

As with other events, you can listen to `MSPointer*` events on whatever elements are relevant to you (remembering again that these are translated into legacy mouse events, so you should not listen to both). The specific events are described as follows, given in the order of their typical sequencing:

- [MSPointerOver](#) Pointer moved into the bounds of the element from outside.

- [MSPointerHover](#) A pointer is hovering over the element.
- [MSPointerDown](#) Pointer down occurred on the element.
- [MSPointerMove](#) Pointer moved across the element.
- [MSPointerUp](#) Pointer was released over the element. (If an element previously captured the touch, it should call its [msReleasePointerCapture](#) method.) Note that if a pointer is moved outside of an element and released, it will receive [MSPointerOut](#) but not [MSPointerUp](#).
- [MSPointerCancel](#) The system canceled a pointer event.
- [MSPointerOut](#) Pointer moved out of the bounds of the element, which also occurs with an up event. This is the last pointer event an element will receive.
- [MSGotPointerCapture](#) The pointer is captured by the element.
- [MSLostPointerCapture](#) The pointer capture has been lost for the element.

These are the names you use with [addEventListener](#); the equivalent property names are of the form [onmspointerdown](#), as usual. It should be obvious that some of these events might not occur with all pointer types—touch screens, for instance, generally don't provide over and hover events, though some that can detect the proximity of a finger are so capable.

The PointerEvents example provided with this chapter's companion content and shown in Figure 9-1 lets you see what's going on with all the mouse, pointer, and gesture events, selectively turning groups of events on and off. (Actually the events are always firing; the checkboxes simply control what is shown in the display.)

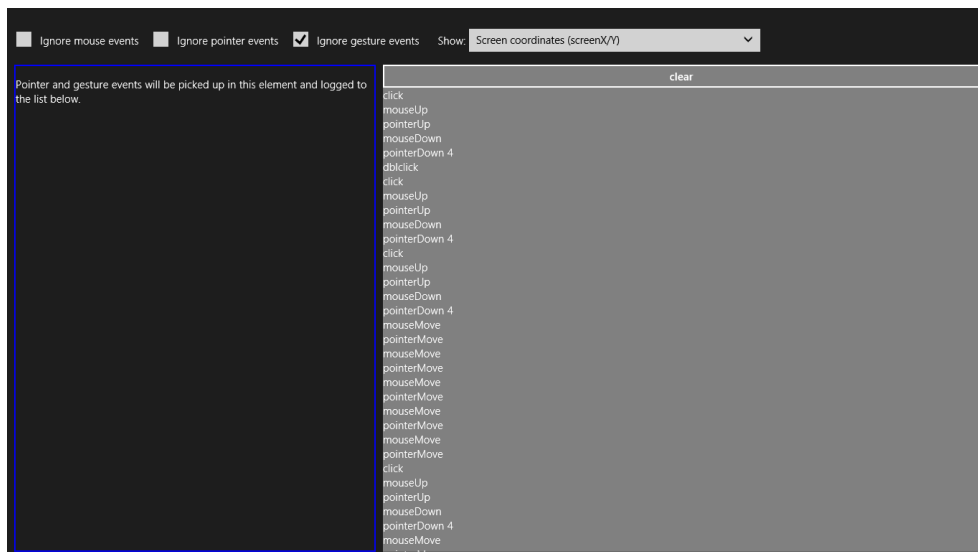


FIGURE 9-1 The PointerEvents example display.

Within the handlers for all of the `MSPointer*` events, the `eventArgs` object contains a whole roster of properties. One of them, `pointerType`, identifies the type of input: touch (2), pen (3), and mouse (4). This property lets you implement different behaviors for different input methods, if desired. For inputs that support more than one simultaneous pointer (as with multitouch), each event object also contains a unique `pointerId` value that identifies a stroke or a path for a specific contact point, allowing you to correlate an initial `MSPointerDown` event with subsequent events. When we look at gestures in the next section, we'll also see how we use the `pointerId` of `MSPointerDown` to associate a gesture with a pointer.

The complete roster of properties that come with the event is actually far too much to show here, as it contains many of the usual [DOM properties](#) along with many pointer-related ones from an object type called `MSPointerEvent`. The best way to see what shows up is to run some code like the [Input DOM pointer and gesture event handling sample](#) (a `canvas` drawing app), set a breakpoint within a handler for one of the events, and examine the event object. Here are some of the more relevant properties to our discussion here:

Properties	Description
<code>currentPoint</code>	A <code>Windows.UI.Input.PointerPoint</code> object. This contains many other properties such as <code>pointerDevice</code> (a <code>Windows.Input.Device.PointerDevice</code> object, as described in "What Input Capabilities Are Present" earlier in this chapter) and one just called <code>properties</code> , which is a <code>Windows.UI.Input.PointerPointProperties</code> .
<code>pointerType</code>	The source of the event could be touch or pen or mouse: <code>MSPOINTER_TYPE_TOUCH</code> (2), <code>MSPOINTER_TYPE_PEN</code> (3), and <code>MSPOINTER_TYPE_MOUSE</code> (4). You can use this to make adjustments according to input type, if necessary.
<code>pointerId</code>	The unique identifier of the contact. This remains the same throughout the lifetime of the pointer.
<code>type</code>	The name of the event, as in "MSPointerDown".
<code>x/screenX, y/screenY</code>	The x- and y-coordinates of the pointer's center point position relative to the screen.
<code>clientX, clientY</code>	The x- and y-coordinates of the pointer's center point position relative to the client area of the app.
<code>offsetX, offsetY</code>	The x- and y-coordinates of the pointer's center point position relative to the element.
<code>button</code>	Determines the button pressed by the user (on mice and other input devices with buttons). The left is 0, middle is 1, and right is 2; these values can be combined with bitwise the OR operator for <i>chord</i> presses (multiple buttons).
<code>ctrlKey, altKey, shiftKey</code>	Indicates whether certain keys were depressed when the pointer event occurred.
<code>hwTimestamp</code>	The timestamp (in milliseconds) at which the event was received from the hardware.
<code>relatedTarget</code>	Provides the element related to the current event, e.g., the <code>MSPointerOut</code> event will provide the element to which the touch is moving. This can be null.
<code>isPrimary</code>	Indicates if this pointer is the primary one in a multitouch scenario (such as the pointer that the mouse would control).
<i>Properties surfaced depending on hardware support (if not supported, these values will be 0)</i>	
<code>width, height</code>	The contact width and height of the touch point specified by <code>pointerId</code> .
<code>pressure</code>	Pen pressure normalized in a range of 0 to 255.
<code>rotation</code>	Clockwise rotation of the cursor around its own major axis in a range of 0 to 359.
<code>tiltX</code>	The left-right tilt away from the normal of a transducer (typically perpendicular to the surface) in a range of -90 (left) to 90 (right).
<code>tiltY</code>	The forward-back tilt away from the normal of a transducer (typically perpendicular to the surface) in a range of -90 (forward/away from user) to 90 (back/toward user).

It's very instructive to run the Input DOM pointer and gesture event handling sample on a

multitouch device, because it tracks each `pointerId` separately allowing you to draw with multiple fingers simultaneously.

Tip If for some reason you want to prevent the translation of an `MSPointer*` event into a legacy mouse event, call the `eventArgs.preventDefault` method within the appropriate event handler.

Pointer Capture

It's common with down and up events for an element to set and release a capture on the pointer. To support these operations, the following methods are available on each element in the DOM and apply to each `pointerId` separately:

Method	Description
<code>msSetPointerCapture</code>	Captures the <code>pointerId</code> for the element so that pointer events come to it and are not raised for other elements (even if you move outside the first element and into another). <code>MSGotPointerCapture</code> will be fired on the element as well.
<code>msReleasePointerCapture</code>	Ends capture, triggering an <code>MSLostPointerCapture</code> event.
<code>msGetPointerCapture</code>	Returns the element with the capture, if any (otherwise <code>null</code>).

We see this in the Input DOM pointer and gesture event handling sample, where it sets capture within its `MSPointerDown` handler and releases it in `MSPointerUp`:

```
this.MSPointerDown = function (evt) {
    canvas.msSetPointerCapture(evt.pointerId);
    // ...
};

this.MSPointerUp = function (evt) {
    canvas.msReleasePointerCapture(evt.pointerId);
    // ...
};
```

Gesture Events

The first thing to know about all `MSGesture*` events is that they don't just fire automatically like `click` and `MSPointer*` events, and you don't just add a listener and be done with it (that's what `click` is for!). Instead, you need to do a little bit of configuration first to tell the system how exactly you want gestures to occur, and you need to use `MSPointerDown` to associate the gesture configurations with a particular `pointerId`. This small added bit of complexity makes it possible for apps to work with multiple concurrent gestures and keep them all independent just as you can do with pointer events. Imagine, for example, a jigsaw puzzle app (as presented in a small way in one of the samples in "The Gesture Samples" below) that allows multiple people sitting around a table-size touch screen to work with individual pieces as they will. Using gestures, each person can be manipulating an individual piece (or two!), moving it around, rotating it, perhaps zooming in to see a larger view, and, of course, testing

out placement. For WinRT apps written in JavaScript, it's also helpful that manipulation deltas for configured elements—which include translation, rotation, and scaling—are given in the coordinate space of the parent element, meaning that it's fairly straightforward to translate the manipulation into CSS transforms and such to make the manipulation visible. In short, there is a great deal of flexibility here when you need it; if you don't, you can use gestures in a simple manner as well. Let's see how it all works.

The first step to receiving gesture events is to create an `MSGesture` object and associate it with the element for which you're interested in receiving events. In the `PointerEvents` example, that element is named `divElement`; you need to store that element in the gesture's `target` property and store the gesture object in the element's `gestureObject` property for use by `MSPointerDown`:

```
var gestureObject = new MSGesture();
gestureObject.target = divElement;
divElement.gestureObject = gestureObject;
```

With this association, you can then just add event listeners as usual. The example shows the full roster of the six gesture events:⁴⁹

```
divElement.addEventListener("MSGestureTap", gestureTap);
divElement.addEventListener("MSGestureHold", gestureHold);

divElement.addEventListener("MSGestureStart", gestureStart);
divElement.addEventListener("MSGestureChange", gestureChange);
divElement.addEventListener("MSGestureEnd", gestureEnd);
divElement.addEventListener("MSInertiaStart", inertiaStart);
```

We're not quite done yet, however. If this is all you do in your code, you still won't receive any of the events because each gesture has to be associated with a pointer. You do this within the `MSPointerDown` event handler:

```
function pointerDown(e) {
    //Associate this pointer with the target's gesture
    e.target.gestureObject.addPointer(e.pointerId);
    e.target.gestureObject.pointerType = e.pointerType;
}
```

If you want to enable rotation and pinch-stretch gestures with the mouse wheel (which you should do), simply add an event handler for the `wheel` event, set the `pointerId` for that event to 1 (a fixed value for the mouse wheel), and send it on to your `MSPointerDown` handler:

```
divElement.addEventListener("wheel", function (e) {
    e.pointerId = 1;    // Fixed pointerId for MouseWheel
    pointerDown(e);
});
```

Now gesture events from both touch and mouse will start to come in *for that element*. (Remember

⁴⁹ In earlier preview releases there were also events named `MSGestureInit` and `MSGestureDoubleTap` that have since been removed.

that mouse wheel by itself is translate, Ctrl+wheel is zoom, and Shift+Ctrl+wheel is rotate.) What's more, if additional `MSPointerDown` events occur for the same element with different `pointerId` values, the `addPointer` method will include that new pointer in the gesture. This automatically enables pinch-stretch and rotation gestures that rely on multiple points.

If you run the `PointerEvents` example (checking `Ignore Mouse Events` and `Ignore Pointer Events`) and start doing taps, tap-holds, and short drags (with touch or mouse), you'll see output like that shown in Figure 9-2.

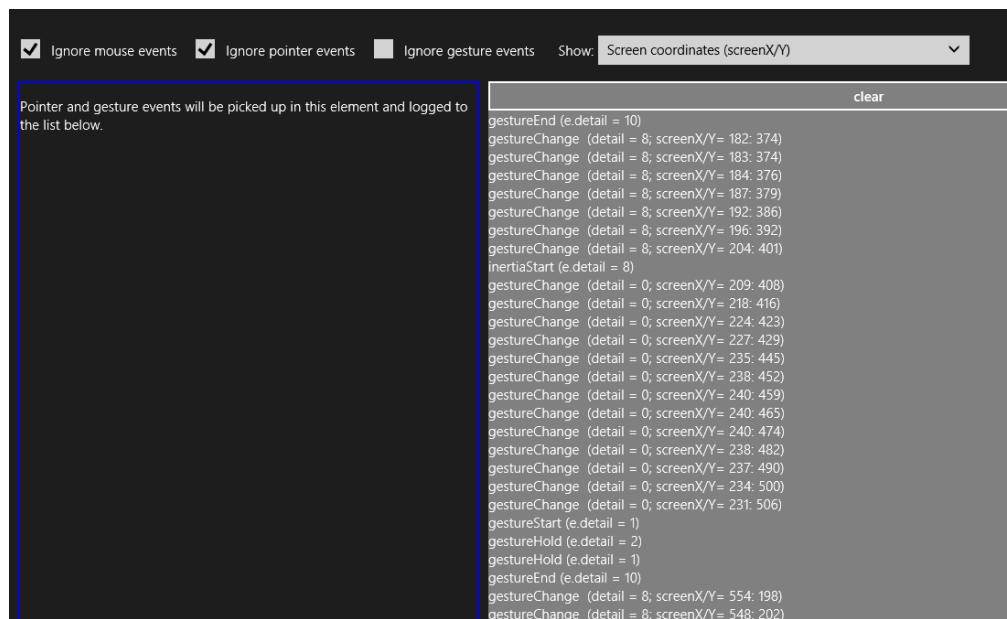
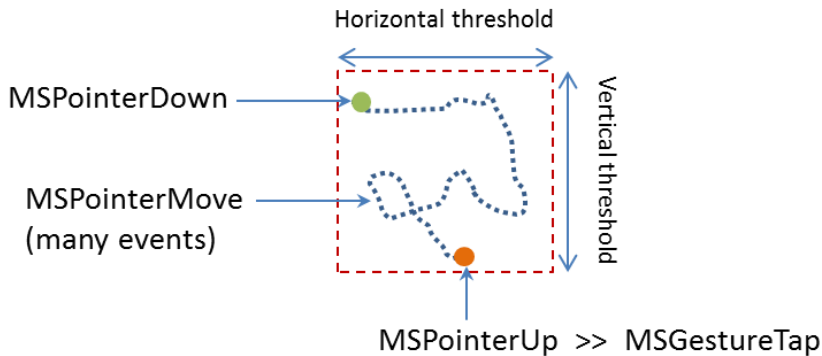


FIGURE 9-2 The `PointerEvents` example output for gesture events (screen shot cropped a bit to emphasize detail).

Again, gesture events are fired in response to a series of pointer events, offering higher-level interpretations of the lower-level pointer events. It's in process of interpretation that differentiates the tap/hold events from the start/change/end events, how and when the `MSInertiaStart` event kicks off, and what the gesture recognizer does when the `MSGesture` object is given multiple points.

Starting with a single pointer gesture, the first aspect of differentiation is a *pointer movement threshold*. When the gesture recognizer sees an `MSPointerDown` event, it starts to watch the `MSPointerMove` events to see whether they stay inside that threshold, which is the effectively boundary for tap and hold events. This accounts for and effectively ignores small amounts of jiggle in a mouse or a touch point as illustrated (or shall I say, exaggerated!) below, where a pointer down, a little movement, and a pointer up generates an `MSGestureTap`:



What then differentiates `MSGestureTap` and `MSGestureHold` is a *time threshold*:

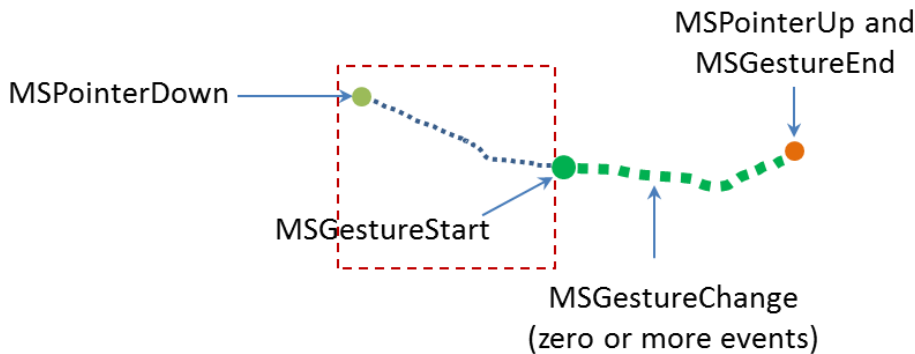
- `MSGestureTap` occurs when `MSPointerDown` is followed by `MSPointerUp` within the time threshold.
- `MSGestureHold` occurs when `MSPointerDown` is followed by `MSPointerUp` outside the time threshold. `MSGestureHold` then fires once when the time threshold is passed with `eventArgs.detail` set to 1 (`MSGESTURE_FLAG_BEGIN`). Provided that the pointer is still within the movement threshold, `MSGestureHold` fires then again when `MSPointerUp` occurs, with `eventArgs.detail` set to 2 (`MSGESTURE_FLAG_END`). You can see this detail included in the first two events of Figure 9-2 above.

The gesture flags in `eventArgs.detail` value is accompanied by many other positional and movement properties in the `eventArgs` object as shown in the following table:

Properties	Description
<code>screenX, screenY</code>	The x- and y-coordinates of the gesture center point relative to the screen.
<code>clientX, clientY</code>	The x- and y-coordinates of the gesture center point relative to the client area of the app.
<code>offsetX, offsetY</code>	The x- and y-coordinates of the gesture center point relative to the element.
<code>translationX, translationY</code>	Translation along the x- and y-axes.
<code>velocityX, velocityY</code>	Velocity of movement along x- and y-axes.
<code>scale</code>	Scale factor for zoom (percentage change in the scale).
<code>expansion</code>	Diameter of the manipulation area (absolute change in size, in pixels).
<code>velocityExpansion</code>	Velocity of expanding manipulation area.
<code>rotation</code>	Rotation angle in radians.
<code>velocityAngular</code>	Angular velocity in radians.
<code>detail</code>	<p>Contains the gesture flags that describe the gesture state of the event; these flags are defined as values in <code>eventArgs</code> itself:</p> <p><code>eventArgs.MSGESTURE_FLAG_NONE</code> (0): Indicates ongoing gesture such as <code>MSGestureChange</code> where there is change in the coordinates.</p> <p><code>eventArgs.MSGESTURE_FLAG_BEGIN</code> (1): The beginning of the gesture sequence. If the interaction contains single event such as <code>MSGestureTap</code>, both <code>MSGESTURE_FLAG_BEGIN</code> and <code>MSGESTURE_FLAG_END</code> flags will be set (detail will be 3).</p> <p><code>eventArgs.MSGESTURE_FLAG_END</code> (2): The end of the gesture sequence. Again, if the interaction contains</p>

	<p>single event such as <code>MSGestureTap</code>, both <code>MSGESTURE_FLAG_BEGIN</code> and <code>MSGESTURE_FLAG_END</code> flags will be set (detail will be 3).</p> <p><code>eventArgs.MSGESTURE_FLAG_CANCEL</code> (4): The gesture was cancelled. Always comes in pair with <code>MSGESTURE_FLAG_END</code>, (detail will be 6).</p> <p><code>eventArgs.MSGESTURE_FLAG_INERTIA</code> (8): The gesture is in an inertia state. The <code>MSGestureChange</code> event can be distinguished from direct interaction and timer driven inertia through this flag.</p>
<code>hwTimestamp</code>	The timestamp of the pointer assigned by the system when the input was received from the hardware.

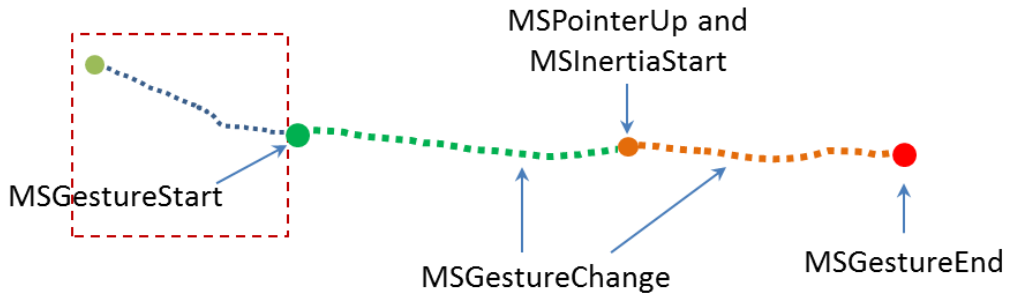
Many of these properties become much more interesting when a pointer moves *outside* the movement threshold, after which time you'll no longer see the tap or hold events. Instead, as soon as the pointer leaves the threshold area, `MSGestureStart` is fired, followed by zero or more `MSGestureChange` events (typically many more!), and completed with a single `MSGestureEnd` event:



Note that if a pointer has been held within the movement threshold long enough for the first `MSGestureHold` to fire with `MSGESTURE_FLAG_BEGIN`, but then the pointer is moved out of the threshold area, `MSGestureHold` will be fired a second time with `MSGESTURE_FLAG_CANCEL | MSGESTURE_FLAG_END` in `eventArgs.detail` (a value of 6), followed by `MSGestureStart` with `MSGESTURE_FLAG_BEGIN`. This series is how you differentiate a hold from a slide or drag gesture even if the user holds the item in place for a while.

Together, the `MSGestureStart`, `MSGestureChange`, and `MSGestureEnd` events define a *manipulation* of the element to which the gesture is attached, where the pointer remains in contact with the element throughout the manipulation. Technically this means that the pointer was no longer moving when it was released.

If the pointer *was* moving when released, then we switch from a manipulation to an *inertial* motion. In this case, an `MSInertiaStart` event gets fired in to indicate that the pointer effectively continues to move even though contact was released or lifted. That is, you'll continue to receive `MSGestureChange` events until the movement is complete:



Conceptually you can see the difference between a manipulation and an inertial motion as illustrated in Figure 9-3; the curves shown here are not necessarily representative of actual changes between messages. If the pointer is moved along the green line such that it's no longer moving when released, we see the series of gesture that define a manipulation. If the pointer is released while moving, we see `MSInertiaStart` in the midst of `MSGestureChange` events and the event sequence follows the orange line.

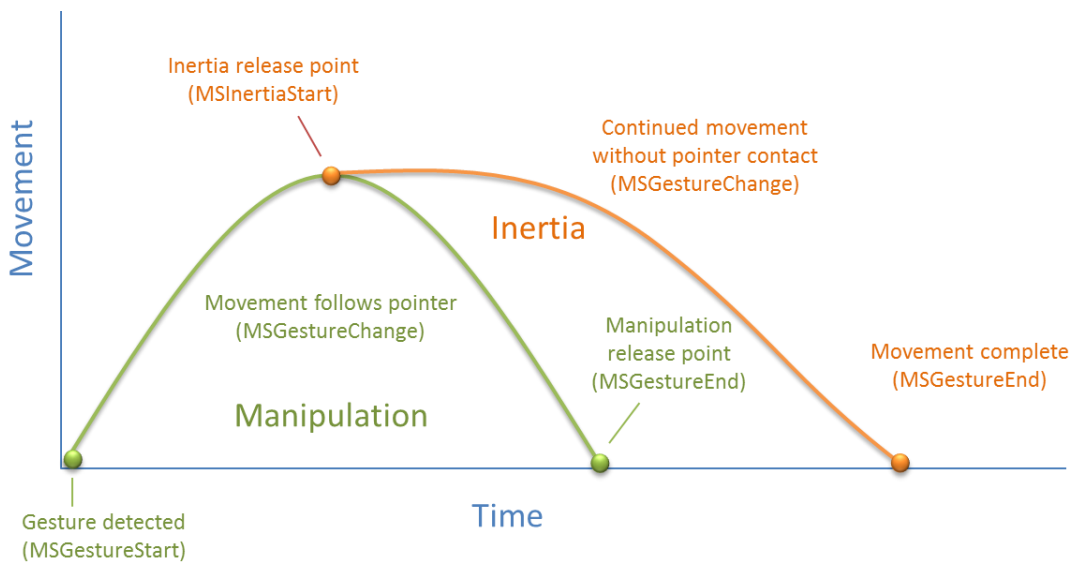


FIGURE 9-3 A conceptual representation of manipulation (green) and inertial (orange) motions.

Referring back to Figure 9-2, when the Show drop-down list (as shown!) is set to Velocity, the output for `MSGestureChange` events includes the `eventArgs.velocity*` values. During a manipulation, the velocity can change at any rate depending on how the pointer is moving. Once an inertial motion begins, however, the velocity will gradually diminish down to zero at which point `MSGestureEnd` occurs. The number of change events depends on how long it takes for the movement to slow down and come to a stop, of course, but if you're just moving an element on the display with these change events, the user will see a nice fluid animation. You can play with this in the PointerEvents example,

using the Show drop-down list to also look at how the other positional properties are affected by different manipulations and inertial gestures.

Multipoint Gestures

What we've discussed so far has focused on a single point gesture, but the same is also true for multipoint gestures. When an `MSGesture` object is given multiple pointers through its `addPointer` event, it will also fire `MSGestureStart`, `MSGestureChange`, `MSGestureEnd` for rotations and pinch-stretch gestures, along with `MSInertiaStart`. In these cases, the `scale`, `rotation`, `velocityAngular`, `expansion`, and `velocityExpansion` properties in the `eventArgs` object become meaningful.

You can selectively view these properties for `MSGestureChange` events through the upper-right drop-down list in the PointerEvents example. One thing you might notice is that if you do multipoint gestures in the Visual Studio simulator, you'll never see `MSGestureTap` events for the individual points. This is because the gesture recognizer can see that multiple `MSPointerDown` events are happening almost simultaneously (which is where the `hwTimestamp` property comes into play) and combines them into an `MSGestureStart` right away (for example, starting a pinch-stretch or rotation gesture).

Now I'm sure you're asking some important questions. While I've been speaking of pinch-stretch, rotation, and translation gestures as different things, how does one, in fact, differentiate these gestures when they're all coming into the app through the same `MSGestureChange` event? Doesn't that just make everything confusing? What's the strategy for translation, rotation, and scaling gestures?

Well, the answer is—you don't have to separate them! If you think about it for a moment, how you handle `MSGestureChange` events and the data each one contains depends on the kinds of manipulations you actually support in your UI:

- If you're supporting only translation of an element, you'll simply never pay any attention to properties like `scale` and `rotation` and apply only those like `clientX` and `clientY`. This would be the expected behavior for selecting an item in a collection control, for example (or a control that allowed drag-and-drop of items to rearrange them).
- If you support only zooming, you'll ignore all the positional properties and work with `scale`, `expansion`, and/or `velocityExpansion`. This would be the sort of behavior you'd expect for a control that supported optical or semantic zoom.
- If you're interested in only rotation, the `rotation` and `velocityAngular` properties are your friends.

Of course, if you want to support multiple kinds of manipulations together, you can simply apply all of these properties together, feeding them into CSS transforms, for instance. This would be expected of an app that allowed arbitrary manipulation of on-screen objects, and it's exactly what the one of the gesture samples of the Windows SDK demonstrates.

The Input Instantiable Gesture Sample

While the `PointerEvents` example included with this chapter gives us a raw view of pointer and gesture events, what really matters to apps is how to apply these events to real manipulation of on-screen objects, which is to say, implementing parts of touch language such as pinch/stretch and rotation. For these we can turn to the [Input Instantiable gestures sample](#).

This sample primarily demonstrates how to use gesture events on multiple elements simultaneously. In Scenarios 1 and 2, the app simulates a simple example of a puzzle app, as mentioned earlier. Each colored box can be manipulated separately, using drag to move (with or without inertia), pinch-stretch gestures to zoom, and rotation gestures to rotate, as shown in Figure 9-4.

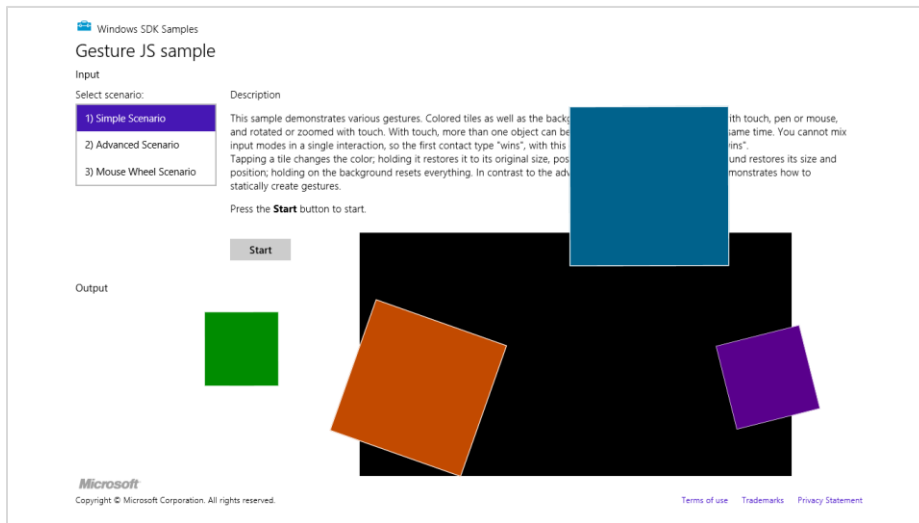


FIGURE 9-4 The Input Instantiable Gestures Sample after playing around a bit. The “instantiable” word comes from the need to instantiate an `MSGesture` object to receive gesture events.

In Scenario 1 (`js/instantiableGesture.js`), an `MSGesture` object is created for each screen element along with one for the black background “table top” element during initialization (the `initialize` function). This is the same as we’ve already seen. Similarly, the `MSPointerDown` handler (`onPointerDown`) adds pointers to the gesture object for each element, adding a little more processing to manage z-index and to add only pointers of the same type to an element. This avoids having simultaneous touch, mouse and stylus pointers working on the same element (which would be odd!):

```
function onPointerDown(e) {
    if (e.target.gesture.pointerType === null) { // First contact
        e.target.gesture.addPointer(e.pointerId); // Attaches pointer to element
        e.target.gesture.pointerType = e.pointerType;
        e.target.freeCapture = 1;
    }
    else if (e.target.gesture.pointerType === e.pointerType) { // Contacts of similar type
        e.target.gesture.addPointer(e.pointerId); // Attaches pointer to element
    }
}
```

```

    }
    // Create a different gesture object if we get a different point type
    else if (e.target.freeCapture === 0) {
        var gObj = new MSGesture();
        gObj.target = e.target;
        gObj.srcElt = e.target;
        e.target.gesture = gObj;
        e.target.gesture.pointerType = e.pointerType;
        e.target.gesture.addPointer(e.pointerId);
        e.target.freeCapture = 1;
    }

    // ZIndex Changes on pointer down. Element on which pointer comes down becomes topmost
    var zOrderCurr = e.target.style.zIndex;
    var elts = document.getElementsByClassName("GestureElement");
    for (var i = 0; i < elts.length; i++) {
        if (elts[i].style.zIndex === 3) {
            elts[i].style.zIndex = zOrderCurr;
        }
        e.target.style.zIndex = 3;
    }
}

```

The `MSGestureChange` handler for each individual piece (`onGestureChange`) then just takes all the translation, rotation, and scaling data in the `eventArgs` object and applies them with CSS. This shows how convenient it is that all those properties are already reported in the coordinate space that we need:

```

function onGestureChange(e) {
    var elt = e.target;
    var m = new MSCSSMatrix(elt.style.msTransform);

    elt.style.msTransform = m.
        translate(e.offsetX, e.offsetY).
        translate(e.translationX, e.translationY).
        rotate(e.rotation * 180 / Math.PI).
        scale(e.scale).
        translate(-e.offsetX, -e.offsetY);
}

```

There's a little more going on in the sample, but what we've shown here are the important parts. Clearly, if you didn't want to support certain kinds of manipulations, you'd again simply ignore certain properties in the event args object.

Scenario 2 of this sample has the same output but is implemented a little differently. As you can see in its `initialize` function (`js/gesture.js`), the only events that are initially registered are for the entire "table top" that contains the black background and a surrounding border. Gesture objects for the individual pieces are created and attached to a pointer within the `MSPointerDown` event (`onTableTopPointerDown`). This approach is much more efficient and scalable to a puzzle app that has hundreds or even thousands of pieces, as gesture objects are held only for as long as a particular piece is being manipulated. Those manipulations are also like those of scenario 1, where all the

[MSGestureChange](#) properties are applied through a CSS transform. For further details, refer to the code comments in `gesture.js`, as they are quite extensive.

Scenario 3 of this sample provides another demonstration of performing translate, pinch-stretch, and rotate gestures using the mouse wheel. As shown in the `PointerEvents` example, the only thing you need to do here is process the `wheel` event, set `eventArgs.pointerId` to 1, and pass that onto your `MSPointerDown` handler that then adds the pointer to the gesture object:

```
elt.addEventListener("wheel", onMouseWheel, false);

function onMouseWheel(e) {
    e.pointerId = 1; // Fixed pointerId for MouseWheel
    onPointerDown(e);
}
```

Again, that's all there is to it. (I love it when it's so simple!) As an exercise, you might try adding this little bit of code to Scenarios 1 and 2 as well.

The Gesture Recognizer

With inertial gestures, which continue to send some number of [MSGestureChange](#) events after pointers are released, you might be asking this question: What, exactly, controls those events? That is, there is obviously a specific deceleration model built into those events, namely the one around which the Windows look and feel is built. But what if you want a different behavior? And what if you want to interpret pointer events in different way altogether?

The agent that interprets pointer events into gesture events is called the gesture recognizer, which you can get to directly through the [Windows.UI.Input.GestureRecognizer](#) object. After instantiating this object with `new`, you then set its [gestureSettings](#) properties for the kinds of manipulations and gestures you're interested in. The documentation for [Windows.UI.Input.GestureSettings](#) gives all the options here, which include `tap`, `doubleTap`, `hold`, `holdWithMouse`, `rightTap`, `drag`, translations, rotations, scaling, inertia motions, and `crossSlide` (swipe). For example, in the [Input manipulations and gestures sample](#) (`ballineight.js`) we can see how it configures a recognizer for tap, rotate, translate, and scale (with inertia):

```
gr = new Windows.UI.Input.GestureRecognizer();

// Configuring GestureRecognizer to detect manipulation rotation, translation, scaling,
// + inertia for those three components of manipulation + the tap gesture
gr.gestureSettings =
    Windows.UI.Input.GestureSettings.manipulationRotate |
    Windows.UI.Input.GestureSettings.manipulationTranslateX |
    Windows.UI.Input.GestureSettings.manipulationTranslateY |
    Windows.UI.Input.GestureSettings.manipulationScale |
    Windows.UI.Input.GestureSettings.manipulationRotateInertia |
    Windows.UI.Input.GestureSettings.manipulationScaleInertia |
    Windows.UI.Input.GestureSettings.manipulationTranslateInertia |
    Windows.UI.Input.GestureSettings.tap;
```

```
// Turn off UI feedback for gestures (we'll still see UI feedback for PointerPoints)
gr.showGestureFeedback = false;
```

The [GestureRecognizer](#) also has a number of properties to configure those specific events. With cross-slides, for example, you can set the [crossSlideThresholds](#), [crossSlideExact](#), and [crossSlideHorizontally](#) properties. You can set the deceleration rates (in pixels/ms²) through [inertiaExpansionDeceleration](#), [inertiaRotationDeceleration](#), and [inertiaTranslationDeceleration](#).

Once configured, you then start passing [MSPointer*](#) events to the recognizer object, specific to its methods named [processDownEvent](#), [processMoveEvents](#), and [processUpEvent](#) (also [processMouseWheelEvent](#), and [processInertia](#), if needed). In response, depending on the configuration, the recognizer will then fire a number of its own events. First, there are discrete events like [crossSliding](#), [dragging](#), [holding](#), [rightTapped](#), and [tapped](#). For all others it will fire a series of [manipulationStarted](#), [manipulationUpdated](#), [manipulationInertiaStarting](#), and [manipulationCompleted](#).

When you're using the recognizer directly, in other words, you'll be listening for [MSPointer*](#) events, feeding them to the recognizer, and then listening for and acting on the recognizer's specific events as above rather than the [MSGesture*](#) events that come out of the default recognizer that is configured by the [MSGesture](#) object.

Again, refer to the documentation on [Windows.UI.Input.GestureRecognizer](#) for all the details and to the sample for some bits of code. As one extra example, here's a snippet to capture a small horizontal motion using the [manipulationTranslateX](#) setting:

```
var recognizer = new Windows.UI.Input.GestureRecognizer();
recognizer.gestureSettings = Windows.UI.Input.GestureSettings.manipulationTranslateX;
var pp = Windows.UI.Input.PointerPoint;
var DELTA = 10;

myElement.addEventListener('MSPointerDown', function (data) {
    recognizer.processDownEvent(pp.getCurrentPoint(data.pointerId));
});
myElement.addEventListener('MSPointerUp', function (data) {
    recognizer.processUpEvent(pp.getCurrentPoint(data.pointerId));
});
myElement.addEventListener('MSPointerMove', function (data) {
    recognizer.processMoveEvents(pp.getIntermediatePoints(data.pointerId));
});

recognizer.addEventListener('manipulationcompleted', function (args) {
    var pt = args.cumulative.translation;
    if (pt.x < -DELTA) {
        // move right
    }
    else if (pt.x > DELTA) {
        // move left
    }
});
```

Beyond the recognizer, do note that you can always go the low-level route and do your own processing of `MSPointer*` events however you want, completely bypassing the gesture recognizer. This would be necessary if the configurations allowed by the recognizer object don't accommodate your specific need. At the same time, now is a good time to re-read "Sidebar: Creating Completely New Gesture?" at the end of the earlier section on the touch language. It addresses a few of the questions about when and if custom gestures are really needed.

Keyboard Input and the Soft Keyboard

After everything to do with touch and other forms of input, it seems almost anticlimactic to consider the humble keyboard. Yet of course the keyboard remains utterly important for textual input, whether it's a physical keyboard or the on-screen "soft" keyboard. It is especially important for accessibility as well, as some users are physically unable to use a mouse or other devices.

Fortunately, there is nothing special about handling input from either keyboard in a WinRT app: simply process `keydown`, `keyup`, and `keypress` events as you already know how to do. This works for both the physical keyboard as well as the soft keyboard.

Case closed? Well, not entirely. There are two special concerns with the soft keyboard: how to make it appear, and the effect of its appearance on app layout. At the end of this section I'll also provide a quick run-down of standard keystrokes for app commands.

Soft Keyboard Appearance and Configuration

The appearance of the soft keyboard happens for one reason and one reason only: the user *touches* a text input element or an element with the `contenteditable="true"` attribute (such as a `div` or `canvas`). There isn't an API to make the keyboard appear, nor will it appear when you click in such an element with the mouse or a stylus or tab to it with a physical keyboard.

The configuration of the keyboard is also sensitive to the type of input control. We can see this through Scenario 2 of the [Input Touch keyboard text input sample](#), where `ScopedViews.html` contains a bunch of `input` controls (surrounding table markup omitted), which appear as shown in Figure 9-5:

```
<input type="url" name="url" id="url" size="50" />
<input type="email" name="email" id="email" size="50" />
<input type="password" name="password" id="password" size="50" />
<input type="text" name="text" id="text" size="50" />
<input type="number" name="number" id="number" />
<input type="search" name="search" id="search" size="50" />
<input type="tel" name="tel" id="tel" size="50" />
```

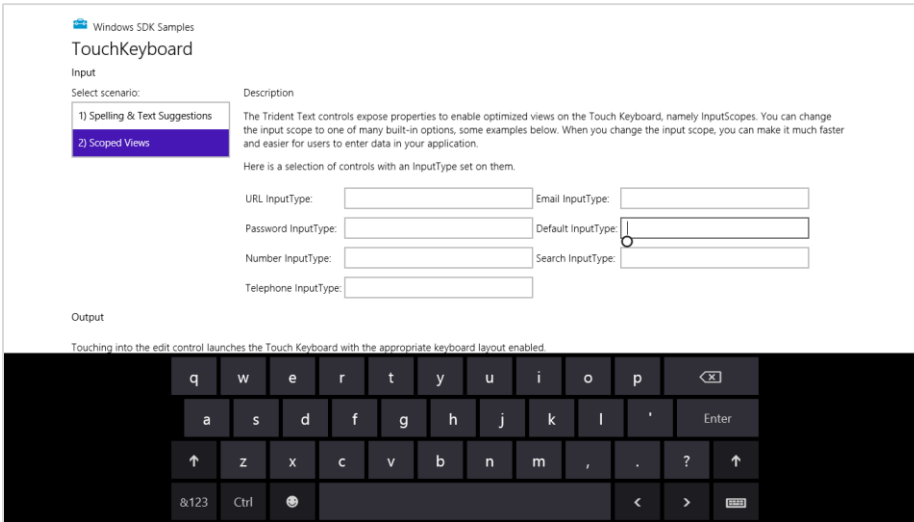


FIGURE 9-5 The soft keyboard appears when you touch an input field, as shown in the Input Touch keyboard text input sample.

What's shown in Figure 9-5 is the default keyboard. If you tap in the Search field, you get pretty much the same view except the Enter key turns into Search. For the Email field, it's much like the default view except you get @ and .com keys to either side of the spacebar:



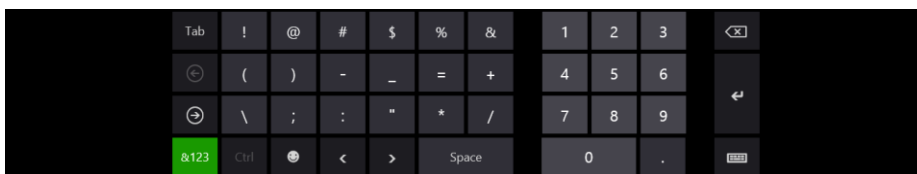
The URL keyboard is the same except the @ key is dropped and Enter turns into Go:



For passwords you get a key to hide keypresses, which prevents a visible animation from happening on the screen—a very important feature if you're recording videos!



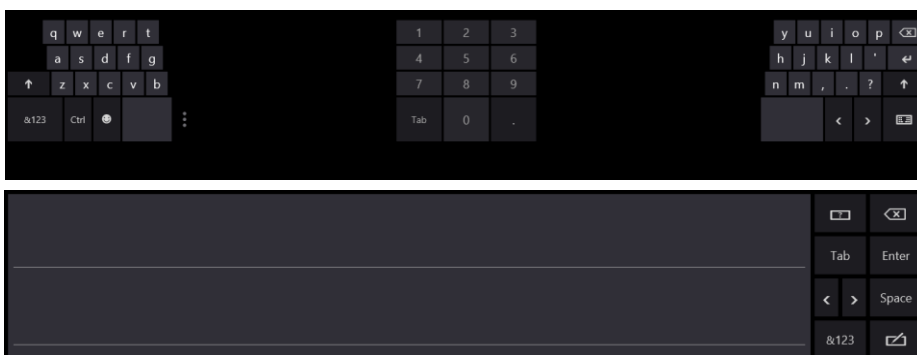
And finally, the Number and Telephone fields bring up a number-oriented view:



In all of these cases, the key on the lower right (whose icon looks a bit like a keyboard), lets you switch to other keyboard layouts:

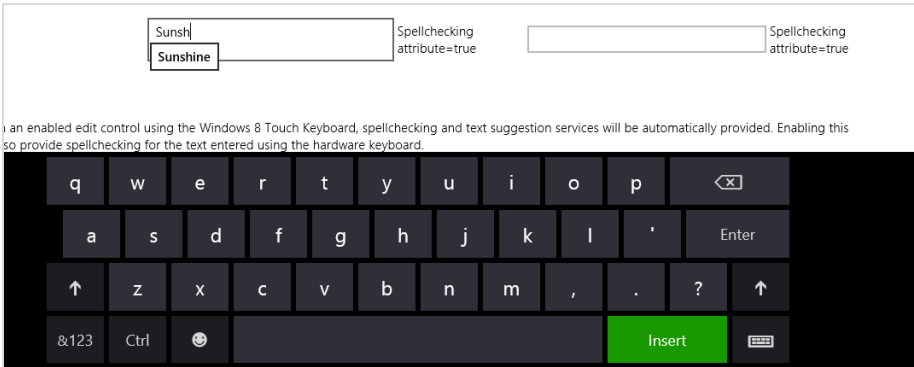


The options here are the normal (wide) keyboard, the split keyboard, a handwriting recognition panel, and a key to dismiss the soft keyboard entirely. Here's what the default split keyboard and handwriting panels look like:



This handwriting panel for input is simply another mode of the soft keyboard: you can switch between the two, and your selection sticks across invocations. (For this reason, Windows does not automatically invoke the handwriting panel for a pen pointer, because the user may prefer to use the soft keyboard even with the stylus.)

The keyboard will also adjust its appearance with text input controls to provide text suggestions; specifically, a highlighted Insert key appears. This is demonstrated in Scenario 1 of the sample and shown below:



Adjusting Layout for the Soft Keyboard

The second concern with the soft keyboard (no, I didn't forget!) is handling layout when it appears because the input field might be positioned such that it would become obscured.

When the soft keyboard or handwriting panel appears, the system will try to make sure the input field is visible by scrolling the page content if it can. This means that it just sets a negative vertical offset to your entire page equal to the height of the soft keyboard. For example, if I add (as a total hack!) a bunch of `
` elements at the top of `ScopedView.html` in the sample such that the input controls are at the bottom of the page, and then I touch one of them, the whole page is slid up, as shown in Figure 9-6.

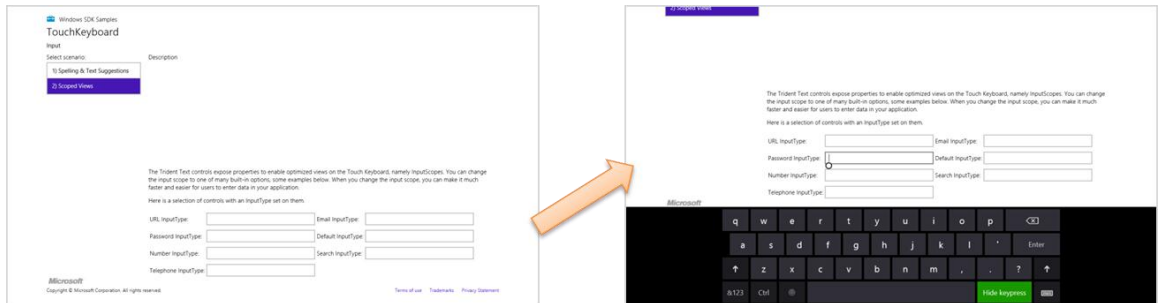


FIGURE 9-6 When the soft keyboard appears, Windows will automatically slide the app page up to make sure the input field isn't obscured.

Although this can be the easiest solution to this particular concern, it's not always ideal. Fortunately, you can do something more intelligent if you'd like by listening to the `hiding` and `showing` events of the `Windows.UI.ViewManagement.InputPane` object and adjust your layout directly. Code for doing this can be found in the—are you ready for this one?—[Responding to the appearance of the on-screen](#)

[keyboard sample](#).⁵⁰ Adding listeners for these events is simple (see the bottom of `js/keyboardPage.js`):

```
var inputPane = Windows.UI.ViewManagement.InputPane.getForCurrentView();
inputPane.addEventListener("showing", showingHandler, false);
inputPane.addEventListener("hiding", hidingHandler, false);
```

Within the `showing` event handler, the `EventArgs.occludedRect` object (a `Windows.Foundation.Rect`) gives you the coordinates and dimensions of the area that the soft keyboard is covering. In response, you can adjust whatever layout properties are applicable and set the `EventArgs.ensuredFocusedElementInView` property to `true`. This tells Windows to bypass its automatic offset behavior:

```
function showingHandler(e) {
    if (document.activeElement.id === "customHandling") {
        keyboardShowing(e.occludedRect);

        // Be careful with this property. Once it has been set, the framework will
        // do nothing to help you keep the focused element in view.
        e.ensuredFocusedElementInView = true;
    }
}
```

The sample will show both cases. If you tap on the aqua-colored `defaultHandling` element on the bottom left of the app, as shown in Figure 9-7, this `showingHandler` does nothing, so the default behavior occurs.

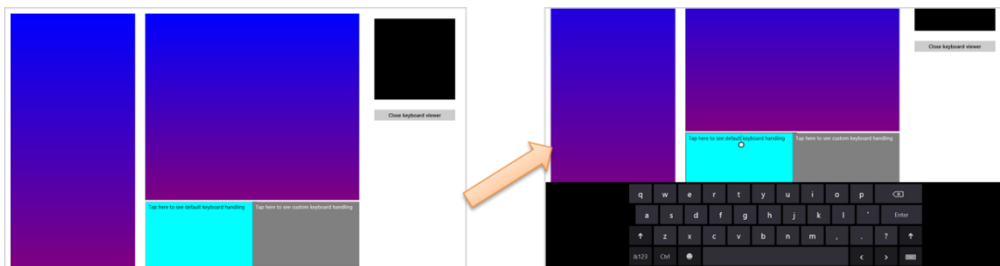


FIGURE 9-7 Tapping on the left `defaultHandling` element at the bottom shows the default behavior when the keyboard appears, which offsets other page content vertically.

If you tap the `customHandling` element (on the right), it calls its `keyboardShowing` routine to do layout adjustment:

```
function keyboardShowing(keyboardRect) {
    // Some code omitted...
```

⁵⁰ And while you might think this is the second longest sample name in the Windows SDK, it actually gets only the bronze medal. The [Unselectable content areas with -ms-user-select CSS attribute sample](#), as we've seen, gets the gold by seven characters. [Using requestAnimationFrame for power efficient animations sample](#) wins the silver by 4. I don't mind such long names, however—I'm delighted that there we have such an extensive set of great samples to draw from!

```

var elementToAnimate = document.getElementById("middleContainer");
var elementToResize = document.getElementById("appView");
var elementToScroll = document.getElementById("middleList");

// Cache the amount things are moved by. It makes the math easier
displacement = keyboardRect.height;
var displacementString = -displacement + "px";

// Figure out what the last visible things in the list are
var bottomOfList = elementToScroll.scrollTop + elementToScroll.clientHeight;

// Animate
showingAnimation = KeyboardEventsSample.Animations.inputPaneShowing(elementToAnimate,
    { top: displacementString, left: "0px" }).then(function () {

    // After animation, layout in a smaller viewport above the keyboard
    elementToResize.style.height = keyboardRect.y + "px";

    // Scroll the list into the right spot so that the list does not appear to scroll
    elementToScroll.scrollTop = bottomOfList - elementToScroll.clientHeight;
    showingAnimation = null;
});
}

```

The code here is a little involved because it's animating the movement of the various page elements. The short of it is that the layout of affected elements—namely the one that is tapped—is adjusted to make space for the keyboard. Other elements on the page are otherwise unaffected. The result is shown in Figure 9-8.

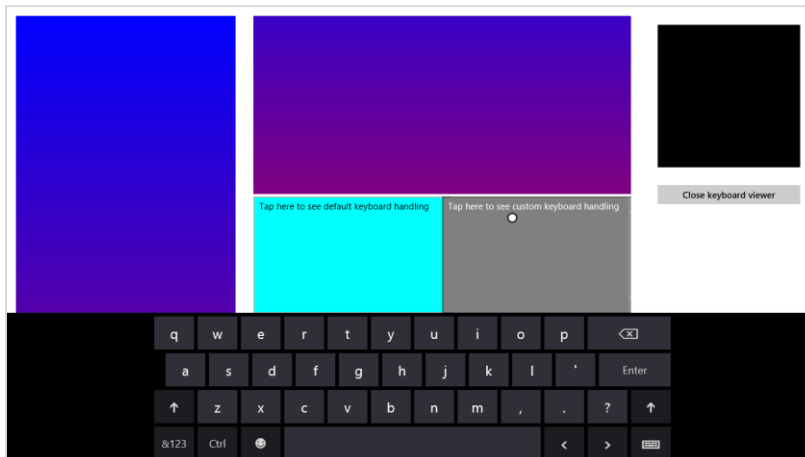


FIGURE 9-8 Tapping the left gray *customHandling* element shows custom handling for the keyboard's appearance.

Standard Keystrokes

The last piece I wanted to include on the subject of the keyboard is a list of command keystrokes you

might support in your app, which are shown in the following table. These are in addition to the touch language equivalents, and you're probably accustomed to using many of them already. They're good to review because again, apps should be fully usable with just the keyboard, and implementing keystrokes like these goes a long way toward fulfilling that requirement and enabling more efficient use of your app by keyboard users.

Action or Command	Keystroke
Move focus	Tab
Back (navigation)	Back button on special keyboards; backspace if not in a text field; Alt+left arrow
Forward (navigation)	Alt+right arrow
Up	Alt+up arrow
Cancel/Escape from mode	ESC
Walk through items in a list	Arrow keys (plus Tab)
Jump through items in a list to next group if selection doesn't automatically follow focus	Ctrl+arrow keys
Zoom (semantic and optical)	Ctrl+ and Ctrl-
Jump to something in a named collection	Start typing
Jump far	Page up/down (should work in panning UI, in either horizontal or vertical directions)
Next tab or group	Ctrl+Tab
Previous tab or group	Ctrl+Shift+Tab
Nth tab or group	Ctrl+N (1-9)
Open app bar	Win+Z
Context menu	Context menu key
Open additional flyout/select menu item	Enter
Navigate into/activate	Enter (on a selection)
Select	Space
Select contiguous	Shift+arrow keys
Pin this	Ctrl+Shift+!
Save	Ctrl+S
Find	Ctrl+F
Print	Ctrl+P (call Windows.Graphics.Printing.PrintManager.showPrintUIAsync)
Copy	Ctrl+C
Cut	Ctrl+X
Paste	Ctrl+V
New Item	Ctrl+N
Open address	Ctrl+L or Alt+D
Rotate	Ctrl+, and Ctrl+.
Play/Pause	Ctrl+P (media apps only)
Next item	Ctrl+F (conflict with Find)
Previous item	Ctrl+B
Rewind	Ctrl+Shift+B
Fast forward	Ctrl+Shift+F

Inking

Beyond the built-in soft keyboard/handwriting pane, an app might also want to provide a surface on which it can directly accept pointer input as *ink*. By this I mean more than just having a [canvas](#) element and processing [MSPointer*](#) events to draw on it to produce a raster bitmap. Ink is a data structure that

maintains the actual input data (including pressure, angle, and velocity if the hardware supports it) which allows for handwriting recognition and other higher-level processing that isn't possible with raster data. Ink, in other words, remembers how an image was drawn, not just the final image itself, and it works with all types of pointer input.

Ink support in WinRT is found in the [Windows.UI.Input.Inking](#) namespace. This API doesn't depend on any particular presentation framework, nor does it provide for rendering: it deals only with the managing data structures that an app can then render itself to a drawing surface such as a [canvas](#). Here's its basic function:

- Create an instance of the manager object with [new Windows.UI.Input.Inking.InkManager](#).
- Assign any drawing attributes by creating a [Windows.UI.Input.Inking.InkDrawingAttributes](#) object and settings attributes like the ink [color](#), [fitToCurve](#) (as opposed to the default straight lines), [ignorePressure](#), [penTip](#) ([Windows.UI.Input.Inking.PenTipShape.circle](#) or [rectangle](#)), and [size](#) (a [Windows.Foundation.Size](#) object with [height](#) and [width](#)).
- For the input element, listen for the [MSPointerDown](#), [MSPointerMove](#), and [MSPointerUp](#) events, which you generally need to handle for display purposes already. The [eventArgs.currentPoint](#) is a [Windows.UI.Input.PointerPoint](#) object that contains a pointer id, point coordinates, and properties like pressure, tilt, and twist.
- Pass that [PointerPoint](#) object to the ink manager's [processPointerDown](#), [processPointerUpdate](#), and [processPointerUp](#) methods, respectively.
- After [processPointerUp](#), the ink manager will create a [Windows.UI.Input.Inking.InkStroke](#) object for that path. Those strokes can then be obtained through the ink manager's [getStrokes](#) method and rendered as desired.
- Higher-order gestures can be also converted into [InkStroke](#) objects directly and given to the manager through its [addStroke](#) method. Stroke objects can also be deleted with [deleteStroke](#).

The ink manager also provides methods for performing handwriting recognition with its contained strokes, saving and loading the data, and handling different modes like draw and erase. For a complete demonstration, check out the [Input Ink sample](#) that is shown in Figure 9-9. This sample lets you see the full extent of inking capabilities, including handwriting recognition.



FIGURE 9-9 The Input Ink sample with many commands on its app bar. The green “Hello” was generated by selecting the Hello ink and tapping the Recognition command.

The SDK also includes the [Input Simplified ink sample](#) to demonstrate a more focused handwriting recognition scenario, as shown in Figure 9-10. You should know that this is one sample that *doesn't* support touch at all—it's strictly mouse and stylus and uses keystrokes for various commands instead of an app bar. Look at the [keydown](#) function in `simpleink.js` for a list of the Ctrl+key commands; the spacebar performs recognition of your strokes and the backspace key clears the canvas. As you can see in the figure, I think the handwriting recognition is quite good! (It tells me that the handwriting samples I gave to an engineering team at Microsoft somewhere in the mid-1990s must have made a valuable contribution.)

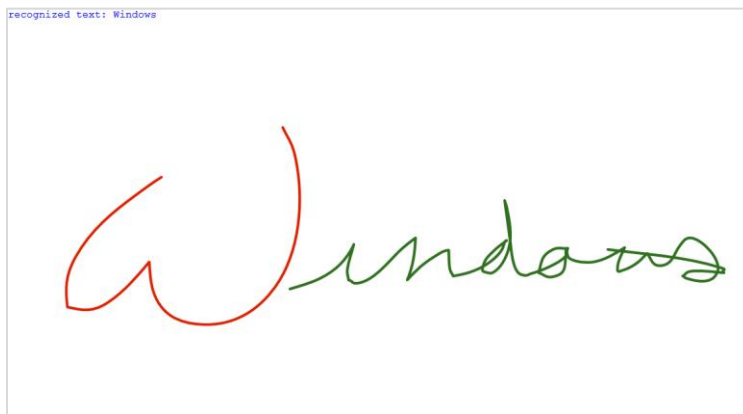


FIGURE 9-10 The Input Simplified Ink sample doing a great job recognizing my sloppy mouse-based handwriting.

Geolocation

Before we explore sensors more generally, I wanted to call out the geolocation capabilities for WinRT apps separately because its API is structured differently from the rest. Indeed, because we've already used this since Chapter 2, "Quickstart" in the Here My Am! App, what we really want to do here is provide the complete story of this highly useful capability.

Unlike all other sensors, in fact, geolocation is the *only* one that has an associated capability you must declare in the manifest. Where you are on the earth is an absolute measure, if you will, and is therefore classified as a piece of personal information. Therefore, users must give their consent before an app can obtain that information, and your app must also provide a Privacy Statement in the Windows Store. Other sensor data, in contrast, is relative—you cannot, for example, really know anything about a *person* from how a device is tilted, how it's moving, or how much light is shining on it. Accordingly, none of those others sensors have a capability you must declare like geolocation: you can use them freely.

As you might know, geolocation can be obtained in two different ways. The primary and most precise way, of course, is to get a reading from an actual GPS radio that is talking to geosynchronous satellites some hundreds of miles up in orbit. The other reasonably useful means, though not always accurate, is to attempt to find one's position through the IP address of a wired network connection or to triangulate from the position of available WiFi hotspots. Whatever the case, WinRT will do its best to give you the best reading it can.

To access geolocation readings, you must first create an instance of the WinRT geolocator, through new [Windows.Devices.Geolocation.Geolocator](#). With that in hand, you can then call its [getGeopositionAsync](#) method, whose results (delivered to your completed handler) is a [Geoposition](#) object (in the same [Windows.Devices.Geolocation](#), as everything here is unless noted). Here's the code as it appears in Here My Am!:

```
var gl = new Windows.Devices.Geolocation.Geolocator();

gl.getGeopositionAsync().done(function (position) {
    //Save for share
    lastPosition = { latitude: position.coordinate.latitude,
        longitude: position.coordinate.longitude };
});
```

The [getGeopositionAsync](#) method also has a [variation](#) where you can specify two parameters: a maximum age for a cached reading (which is to say, how stale you can allow a reading to be) and a timeout value for how long you're willing to wait for a response. Both values are in milliseconds.

A [Geoposition](#) contains two properties. First, its [coordinate](#) property is a [Geocoordinate](#) object that provides [accuracy](#) (meters), [altitude](#) (meters), [altitudeAccuracy](#) (meters), [heading](#) (degrees relative to true north), [latitude](#) (degrees), [longitude](#) (degrees), [speed](#) (meters/sec), and a [timestamp](#) (Date). The second part of a [Geoposition](#) is a [CivicAddress](#) object named—what else!—[civicAddress](#), which might contain [city](#) (string), [country](#) (string, a two-letter ISO-3166 country code), [postalCode](#) (string),

`state` (string), and `timestamp` (Date) properties, if the geolocation provider supplies such data.⁵¹

You can indicate the accuracy you're looking for through the Geolocator's `desiredAccuracy` property, which is either `PositionAccuracy.default` or `PositionAccuracy.high`. The latter, mind you, will be much more radio or network intensive. This might incur higher costs on metered broadband connections and can shorten battery life, so set this to `high` only if it's essential to your user experience.

The Geolocator also provides a `locationStatus` property, which is a `PositionStatus` object containing `ready`, `initializing`, `noData`, `disabled`, `notInitialized`, and `notAvailable`. It should be obvious that you can't get data from a Geolocator that's in any state other than `ready`. To track this, you can listen to the Geolocator's `statusChanged` event, where `eventArgs.status` property in your handler will contain a `PositionStatus`; this is helpful when you find that a GPS device might take a couple seconds to provide a reading. For an example of using this event, see Scenario 1 of the [Geolocation sample](#) in the Windows SDK:

```
geolocator = new Windows.Devices.Geolocation.Geolocator();
geolocator.addEventListener("statuschanged", onStatusChanged);

function onStatusChanged(e) {
    switch (e.status) {
        // ...
    }
}
```

Note that `PositionStatus` and `statusChanged` reflect the readiness of the GPS *device*, and that readiness is not affected by the Location permission in an app's the Settings pane. As demonstrated in *Here My Am!*, an app needs to check permissions by trying to obtain a setting, which is a different concern from device readiness.

The other two interesting properties of the Geolocator are `movementThreshold`, a distance in meters that the device can move before another reading is triggered (which can be used for geo-fencing scenarios), and `reportInterval`, which is the number of milliseconds between attempted readings. Be conservative with the latter, setting it to what you really need, because you again want to minimize network or radio activity. In any case, when the Geolocator takes another reading and finds that the device has moved beyond the `movementThreshold`, it will fire a `positionChanged` event, where the `eventArgs.position` property is a new `Geoposition` object. This is also shown in Scenario 1 of the Geolocation sample:

```
geolocator.addEventListener("positionchanged", onPositionChanged);

function onPositionChanged(e) {
    var coord = e.position.coordinate;
```

⁵¹ That is, the `civicAddress` property might not be available or might be empty. An alternate means to obtain it is to use the [Bing Maps API](#) to convert coordinates into an address.

```

document.getElementById("latitude").innerHTML = coord.latitude;
document.getElementById("longitude").innerHTML = coord.longitude;
document.getElementById("accuracy").innerHTML = coord.accuracy;
}

```

With `movementThreshold` and `reportInterval`, really think through what your app needs based on the accuracy and/or refresh intervals of the data you're using in relation to the location. For example, weather data is regional and might be updated only hourly. Therefore, `movementThreshold` might be set on the scale of miles or kilometers and `reportInterval` at 15, 30, 60 minutes, or longer. A mapping or real-time traffic app, on the other hand, works with data that is very location-sensitive and will thus have a much smaller threshold and a much shorter interval.

Where battery life is concerned, it's best to simply take a reading when the user wants one, rather than following the position at regular intervals. But this again depends on the app scenario, and you could also provide a setting that lets the user control geolocation activity.

It's also very important to note that apps won't get `positionChanged` or `statusChanged` events while suspended unless you register a time trigger background task for this purpose and the user adds the app to the lock screen. We'll talk more of this in Chapter 13, "Tiles, Notifications, the Lock Screen, and Background Tasks," and you can also see how this works in Scenario 3 of the Geolocation sample. If, however, you don't use a background task or the user doesn't place you on the lock screen and you still want to track the user's position, be sure to handle the `resuming` event and refresh the position there.

On the flip side, some geolocation scenarios, such as providing navigation, need to also keep the display active (preventing automatic screen shutoff) even when there's no user activity. For this purpose you can use the [Windows.System.Display.DisplayRequest](#) class, namely its `requestActive` and `releaseRelease` methods that you would call when starting and ending a navigation session. Of course, since keeping the display active consumes more battery power, only use this capability when necessary—as when specifically providing navigation—and avoid simply making the request when your app starts. Otherwise your app will probably gain a reputation in the Windows Store as being power hungry!

Sidebar: HTML5 Geolocation

An experienced HTML/JavaScript developer might wonder why WinRT provides a Geolocation API when HTML5 already has one: `window.navigator.geolocation` and its `getCurrentPosition` method that returns an object with coordinates. The reason for the overlap is that other languages like C#, Visual Basic, and C++ don't have another API to draw from, which leaves HTML/JavaScript developers a choice. Under the covers, the HTML5 API hooks into the same data as the WinRT API, requires the same manifest capability, and is subject to the same user consent, so for the most part the two APIs are almost equivalent. I would give WinRT a slight edge due to the `movementThreshold` option, which helps an app cooperate with power management and enables easier geo-fencing. Doing the same with the HTML5 API would require more frequent polling and battery consumption. For many scenarios, however, you can

use either one with equal results.

Like all other WinRT APIs, however, [Windows.Devices.Geolocation](#) is available only in local context pages in a WinRT app; within web context pages you can use the HTML5 API.

Sensors

As I wrote in the introduction to this chapter, I like to think of sensors as another form of input. It actually makes a lot of sense because every device that is now wholly integrated into our computer systems—such that we take them for granted—was at one point a kind of human-interface peripheral. In time, I suspect that many of the sensors that are new to us today will be standard equipment just about everywhere.

Sensors, again, are a way of understanding the relationship of a device to the physical world around it, and this constitutes input because you, as a human being, can affect that relationship primarily by moving the device around in physical space or otherwise changing its environment. Sensors can also be used as direct input to cause motion on the screen rather than relying on some form of abstract input like the keyboard or mouse. For example, instead of using keystrokes to abstractly tilt a game board, you can, with sensors, just tilt the device. Shaking, in fact, is becoming a well-known physical gesture that can be wired to a command of some kind like *Retry Now, darn you! Why aren't you doing what I want?* Haven't we for years been shaking or smacking our computers when they aren't behaving properly? Well, with sensors the computer can now actually respond!

Here, then, is what the various sensors tell us:

- **Location** The device's position on the earth (as we covered in the previous section).
- **Compass and orientation** The direction the device is pointing, relative to the earth's magnetic poles or relative to the device's inherent sense of position (both simple and complex orientation).
- **Inclinometer** The static pitch, roll, and yaw of the device in 3D space.
- **Gyrometer** The angular velocity/rotational motion of the device in 3D space.
- **Accelerometer** The linear G-force acceleration of the device within 3D space (x, y, z).
- **Ambient light** The amount of light surrounding the device.

These are the sensors that are represented in the WinRT API,⁵² some of which are created in

⁵² There is also the proximity sensor for near-field communications (NFC) that tells us when devices are near one another or make contact, but this is more a networking handshake than a sensor like the others. We'll see this in Chapter 14, "Networking."

software through *sensor fusion*. This means taking raw data from one or more hardware sensors and combining, interpreting, and presenting it all in a form that's more directly useful to apps. Just as with pointers, you can still get to raw data if you want it, but oftentimes it's unnecessary. For example, the Simple Orientation sensor provides a simple interpretation of how the device is oriented in relation to its default position, rounding everything off, as it were, to the nearest 90-degree quadrant. The full Orientation sensor, on the other hand, combines gyrometer, accelerometer, and compass data to provide an exact 3D orientation matrix that is much more precise but much more oriented (if I might make the pun!) to advanced scenarios than simply needing to know whether the device is upside-down or rightside-up.

Because all of these sensors are very similar in how they work (which is intentional, with the exception of the Simple Orientation sensor, which is intentionally dissimilar!), I want to show the general pattern of the sensor APIs rather than explicit examples for each. Such examples are readily available in these SDK samples: [Accelerometer](#), [Compass](#), [Gyrometer](#), [Inclinometer](#), [Light Sensor](#), and [OrientationSensor](#)

The usage pattern is as follows, with the particulars summarized in the table that follows:

- Obtain a sensor object via `Windows.Devices.Sensors.<sensor>.getDefault()`.
- Call that object's `getCurrentReading` to obtain a one-time reading.
- For ongoing readings, configure the object's `minimumReportInterval` and `reportInterval` properties (both in milliseconds) and listen to the object's `readingchanged` event. Your handler will receive a reading object of an appropriate type in response. As with geolocation, setting these values wisely will help optimize battery life by avoiding excess electrons flying through the sensors!

Sensor Name (Windows.Devices.Sensors.)	Added Members	Reading Type (Windows.Devices.Sensors)	Reading Properties (timestamp is a Date; all others are Numbers unless noted)
Accelerometer	Event: <code>shaken</code> (event args contains only a <code>timestamp</code> property)	AccelerometerReading	<code>accelerationX</code> (G's), <code>accelerationY</code> , <code>accelerationZ</code> , <code>timestamp</code>
Compass	n/a	CompassReading	<code>headingMagneticNorth</code> (degrees), <code>headingTrueNorth</code> , <code>timestamp</code>
Gyrometer	n/a	GyrometerReading	<code>angularVelocityX</code> (degrees/sec), <code>angularVelocityY</code> , <code>angularVelocityZ</code> , <code>timestamp</code>
Inclinometer	n/a	InclinometerReading	<code>pitchDegrees</code> (degrees), <code>rollDegrees</code> (degrees), <code>yawDegrees</code> (degrees), <code>timestamp</code>
LightSensor	n/a	LightSensorReading	<code>illuminanceInLux</code> (lux), <code>timestamp</code>
OrientationSensor	n/a	OrientationSensorReading	<code>quaternion</code> , (<code>SensorQuaternion</code> containing <code>w</code> , <code>x</code> , <code>y</code> , and <code>z</code> properties) <code>rotationMatrix</code> (<code>SensorRotationMatrix</code> containing <code>m11</code> , <code>m12</code> , <code>m13</code> , <code>m21</code> , <code>m22</code> , <code>m23</code> , <code>m31</code> , <code>m32</code> , <code>m33</code> properties), <code>timestamp</code>

Here's an example of such code from the Gyrometer sample (js/scenario1.js):

```
gyrometer = Windows.Devices.Sensors.Gyrometer.getDefault();

var minimumReportInterval = gyrometer.minimumReportInterval;
var reportInterval = minimumReportInterval > 16 ? minimumReportInterval : 16;
gyrometer.reportInterval = reportInterval;

gyrometer.addEventListener("readingchanged", onDataChanged);

function onDataChanged(e) {
    var reading = e.reading;

    document.getElementById("eventOutputX").innerHTML = reading.angularVelocityX.toFixed(2);
    document.getElementById("eventOutputY").innerHTML = reading.angularVelocityY.toFixed(2);
    document.getElementById("eventOutputZ").innerHTML = reading.angularVelocityZ.toFixed(2);
}
```

With the Orientation Sensor, a *quaternion* can be most easily understood as a rotation of a point [x,y,z] about a single arbitrary axis. This is different from a rotation matrix, which represents rotations around three axes. The mathematics behind quaternions is fairly exotic because it involves the geometric properties of complex numbers and mathematical properties of imaginary numbers, but working with them is simple and frameworks like DirectX support them. See the OrientationSensor sample for more.

Speaking of orientation, I'd mentioned that the [SimpleOrientationSensor](#) works a little differently. Its purpose is to supply *quadrant* orientation rather than exact orientation, which is perhaps all you need. For example, a star chart app would need to know if a slate device is upside-down so that it can adjust its display (along with a compass reading) to match the sky itself.

To summarize this sensor's usage:

- Call `Windows.Devices.Sensors.SimpleOrientation.getDefault` to obtain the object.
- Call the `getCurrentOrientation` to obtain a reading.
- The `orientationChanged` event provides for ongoing readings, where `eventArgs` contains `orientation` (a reading) and `timestamp` properties.
- The reading is a [SimpleOrientation](#) object that contains these properties:
 - `notRotated` ("portrait up"), `rotated90DegreesCounterclockwise` ("portrait left"), `rotated90DegreesCounterclockwise` ("portrait down"), `rotated270DegreesCounterclockwise` ("landscape right") Note that these are entirely different from view states like fullscreen-landscape and fullscreen-portrait.
 - `faceup`, `facedown` (slate devices only).

For a demonstration, see the [SimpleOrientationSensor sample](#).

What We've Just Learned

- “Design for touch, get mouse and stylus for free” is a message that holds true, because working with pointer and gesture input from a variety of input devices doesn’t require you to differentiate between the forms of input.
- Using built-in controls is the easiest way to handle input, but you can also handle `MSPointer*` events and `MSGesture*` events directly, when needed. You can also feed `MSPointer*` events into a custom gesture recognizer (that issues its own events).
- The Windows 8 touch language includes tap, press and hold, slide/pan, cross-slide (to select), pinch-stretch, rotate, and edge gestures (from top/bottom and from the sides). A tap is typically handled with a click `event`, whereas the others require the creation of an `MSGesture` object, association of that object with a pointer, and handling of `MSGesture*` event sequences which provide for manipulations and inertial motions together.
- The touch language also has mouse, stylus, and keyboard equivalents. For mouse and stylus, there is very little work an app needs to do (such as sending mouse `wheel` events to the gesture object). Keyboard support must be implemented separately, but simply uses the standard HTML/JavaScript events.
- Keyboard support also includes accommodating the soft (on-screen) keyboard, which appears automatically for text input fields and other content-editable elements. It automatically adjusts its appearance according to input type, and will pan the app contents up if necessary to avoid having the keyboard overlap the input control. An app can also handle visibility events directly to provide a better experience than the default.
- The Inking API provides apps with the means to record, save, and render an entire series of pointer activities, where the strokes can also be fed into a handwriting recognizer.
- The Geolocation API in WinRT, similar to the HTML5 geolocation API, provides apps with access to GPS data as well as events when the device has moved past a specified threshold.
- The WinRT API represents a number of sensors that can also be used as input to an app. In addition to geolocation, the sensors are compass, orientation, simple orientation (quadrant-based), inclinometer, gyrometer, accelerometer, and ambient light.
- Most sensors follow the same usage pattern: acquire the sensor object, get a current reading, and possibly listen to the `readingchanged` event. They are very easy to work with, leaving much of your energy to apply them creatively!

Chapter 10

Media

To say that media is important to apps—and to culture in general—is a gross understatement. Ever since the likes of Edison made it possible to record a performance for later enjoyment, and the likes of Marconi made it possible to widely broadcast and distribute such performances, humanity’s worldwide appetite for media—graphics, audio, and video—has probably outpaced the appetite for automobiles, electricity, and even junk food. In the early days of the Internet, graphics and images easily accounted for the bulk of network traffic. Today, streaming video even from a single source like Netflix holds top honors for pushing the capabilities of our broadband infrastructure! (It certainly holds true in my own household with my young son’s love of *Curious George* and *Bob the Builder*.)

Incorporating some form of media is likely a central concern for most WinRT apps. Simple ones, even, probably use at least a few graphics to brand the app and present an attractive UI, as we’ve already seen on a number of occasions. Many others, especially games, will certainly use graphics, video, and audio together. In the context of this book, all of this means using the `img`, `svg` (Scalable Vector Graphics), `canvas`, `audio`, and `video` elements of HTML5.

Of course, working with media goes well beyond just presentation because apps might also provide any of the following capabilities:

- Organize and edit media files, including those in the pictures, music, and videos media libraries.
- Transcode (convert) media files, possibly applying various filters and custom codecs.
- Organize and edit playlists.
- Capture audio and video from system devices.
- Stream media from a server to a device, or from a device to a PlayTo target, perhaps also applying DRM.

These capabilities, for which many WinRT APIs exist, along with the media elements of HTML5 and their particular capabilities within the Windows 8 environment, will be our focus for this chapter.

Note As is relevant to this chapter, a complete list of audio and video formats that are supported for WinRT apps can be found on [Supported audio and video formats](#).

Sidebar: Performance Tricks for Faster Apps

Some of the recommendations in this chapter come from a great talk by Jason Weber, the Performance Lead for Internet Explorer, called [50 Performance Tricks to Make Your Windows 8](#)

[Apps Using HTML5 Faster](#). While some of these tricks are specifically for web applications running in a browser, many of them are wholly applicable to WinRT apps written in JavaScript as they run on top of the same infrastructure as Internet Explorer.

Creating Media Elements

Certainly the easiest means to incorporate media into an app is what we've already been doing for years: simply use the appropriate HTML element in your layout and *voilà!* there you have it. With `img`, `audio`, and `video` elements, in fact, you're completely free to use content from just about any location. That is, the `src` attributes of these elements can be assigned URLs that point to in-package content (using relative paths or paths based on `Windows.ApplicationModel.Package.current.installedLocation` that you then pass to `URL.createObjectURL`), files in your app data folders (using `ms-appx://` URLs or paths based on `Windows.Storage.ApplicationData.current` again using `URL.createObjectURL`), and remote files with `http://` and other URLs. With the `img` element, this includes using SVG files as the source.

There are three ways to create a media element in a page or page control.

First is to include the element directly in declarative HTML. Here it's often useful to use the `preload="auto"` attribute for remote audio and video to increase the responsiveness of controls and other UI that depend on those elements. (Doing so isn't really important for local media files since they are, well, already local!) Oftentimes, media elements are placed near the top of the HTML file, in order of priority, so that downloading can begin while the rest of the document is being parsed.

On the flip side, if the user can wait a short time to start a video, use a preview image in place of the video and don't start the download until it's actually necessary. Code for this is shown later in this chapter in the "Video Playback and Deferred Loading" section.

Playback for a declarative element can be automatically started with the `autoplay` attribute, though the built-in UI if the element has the `controls` attribute, or by calling `<element>.play()` from JavaScript.

The second method is to create an HTML element in JavaScript via `document.createElement` and add it to the DOM with `<parent>.appendChild` and similar methods. Here's an example (using media files that are included with this chapter's companion content, though you'll need to drop the code into a new project of your own):

```
//Create elements and add to DOM, which will trigger layout
var picture = document.createElement("img");
picture.src = "media/wildflowers.jpg";
picture.width = 300;
picture.height = 450;
document.getElementById("divShow").appendChild(picture);

var movie = document.createElement("video");
```



```

movie.src = "media/ModelRocket1.mp4";
movie.autoplay = false;
movie.controls = true;
document.getElementById("divShow").appendChild(movie);

var sound = document.createElement("audio");
sound.src = "media/SpringyBoing.mp3";
sound.autoplay = true; //Play as soon as element is added to DOM
sound.controls = true; //If false, audio plays but does not affect layout

document.getElementById("divShow").appendChild(sound);

```

Unless otherwise hidden by styles, image and video elements, plus audio elements with the `controls` attribute, will trigger re-rendering of the document layout. An audio element *without* that attribute will not cause re-rendering.

As with declarative HTML, setting `autoplay` to `true` will cause video and audio to start playing as soon as the element is added to the DOM.

Finally, for audio, apps can also create an `Audio` object in JavaScript to play sounds or music without any effect on UI. More on this later. JavaScript also has object classes for `Image`, and the `Audio` class can be used to load video:

```

//Create objects (pre-loading), then set other DOM object sources accordingly
var picture = new Image(300, 450);
picture.src = "http://www.kraigbrockschmidt.com/downloads/media/wildflowers.jpg";
document.getElementById("image1").src = picture.src;

//Audio object can be used to pre-load (but not render) video
var movie = new Audio("http://www.kraigbrockschmidt.com/downloads/media/ModelRocket1.mp4");
document.getElementById("video1").src = movie.src;

var sound = new Audio("http://www.kraigbrockschmidt.com/downloads/media/SpringyBoing.mp3");
document.getElementById("audio1").src = sound.src;

```

Creating an `Image` or `Audio` object from code does not create elements in the DOM, which can be a useful trait. The `Image` object, for instance, has been used for years to preload an array of image sources for use with things like image rotators and popup menus. Preloading in this case only means that the images have been downloaded and cached. This way, assigning the same URL to the `src` attribute of an element that *is* in the DOM, as shown above, will have that image appear immediately. The same is true for preloading video and audio, but again, this is primarily helpful with remote media as files on the local file system will load relatively quickly as-is. Still, if you have large local images and want them to appear quickly when needed, preloading them into memory is a useful strategy.

Of course, you might want to load media only when it's needed, in which case the same type of code can be used with existing elements, or you can just create an element and add it to the DOM as shown earlier.

Graphics Elements: Img, Svg, and Canvas (and a Little CSS)

I know you're probably excited to get to sections of this chapter on video and audio, but we cannot forget that images have been the backbone of web applications since the beginning and remain a huge part of any app's user experience. Indeed, it's helpful to remember that video itself is conceptually just a series of static images sequenced over time! Fortunately, HTML5 has greatly expanded an app's ability to incorporate image data by adding SVG support and the `canvas` element to the tried-and-true `img` element. Furthermore, applying CSS animations and transitions (covered in detail in Chapter 11, "Purposeful Animations") to otherwise static image elements can make them appear very dynamic.

Speaking of CSS, it's worth noting that many graphical effects that once required the use of static images can be achieved with *just* CSS, especially CSS3:

- Borders, background colors, and background images
- Folder tabs, menus, and toolbars
- Rounded border corners, multiple backgrounds/borders, and image borders
- Transparency
- Embeddable fonts
- Box shadows
- Text shadows
- Gradients

In short, if you've ever used `img` elements to create small visual effects, create gradient backgrounds, use a nonstandard font, or provide some kind of graphical navigation structure, there's probably a way to do it in CSS. For details, see the great [overview of CSS3](#) by Smashing Magazine as well as the CSS specs at <http://w3c.org>. CSS also provides the ability to declaratively handle some events and state using pseudo-selectors of `hover`, `visited`, `active`, `focus`, `target`, `enabled`, `disabled`, and `checked`. For more, see <http://css-tricks.com/> as well as another Smashing Magazine [tutorial on pseudo-classes](#).

That said, let's review the three primary HTML5 elements for graphics:

- `img` is used for raster data. The PNG format generally preferred over other formats, especially for text and line art. GIF is generally considered outdated, as the primary scenarios where GIF produced a smaller file size can probably be achieved with CSS directly. Where scaling is concerned, WinRT apps need to consider resolution scaling, as we saw in Chapter 6, "Layout," and provide separate image files for each scale the app might encounter.

- SVGs are best used for smooth scaling across display sizes and resolution scales. SVGs can be declared inline, created dynamically in the DOM, or maintained as separate files and used as a source for an `img` element (in which case all the scaling characteristics are maintained). An `svg` file can also be used for an `iframe` source, which has the added benefit that the SVG's child elements are accessible in the DOM. As we saw in Chapter 6, preserving the aspect ratio of an SVG is often important, for which you employ the `viewBox` and `preserveAspectRatio` attributes of the `svg` tag.
- The `canvas` element provides a drawing surface for WinRT apps, which is to say an API for creating graphics with lines, rectangles, arcs, text, and so forth. The canvas ultimately generates raster data, which means that once created, a canvas scales like a bitmap. (An app, of course, will typically redraw a canvas with scaled coordinates when necessary to avoid pixelation.) The canvas is also very useful for performing pixel manipulation, even on individual frames of a video while it's playing.

Apps often use all three of these elements to draw on their various strengths. I say this because when `canvas` first became available, developers seemed so enamored with it that they seemed to forget how to use `img` elements, and they ignored the fact that SVGs are often a better choice altogether! (And did I already say that CSS can accomplish a great deal by itself as well?)

In the end, it's helpful to think of all the HTML5 graphics elements as ultimately producing a bitmap that the app host simply renders to the display. You can, of course, programmatically animate the internal contents of these elements in JavaScript, as we'll see in Chapter 11, but for our purposes here it's helpful to simply think of these as essentially static.

What differs between the elements is how image data gets into the element to begin with. `Img` elements are loaded from a source file, `svg`'s are defined in markup, and `canvas` elements are filled through procedural code. But in the end, as the example below demonstrates (Scenario 1 in the HTML Graphics example for this chapter), each can produce identical results, as shown in Figure 10-1.

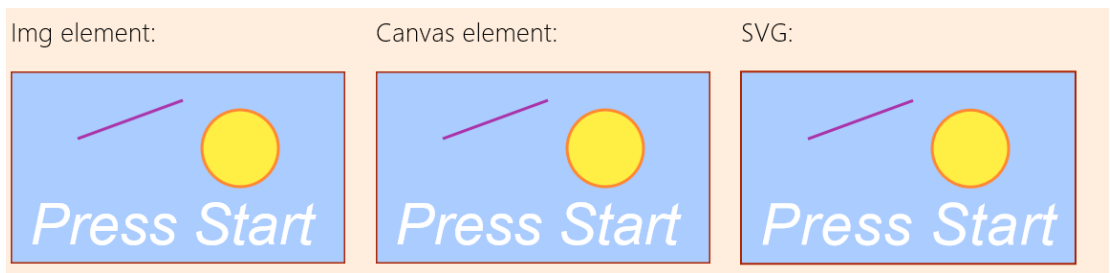


FIGURE 10-1 Image, canvas, and `svg` elements showing identical results.

In short, there are no fundamental differences as to what *can* be rendered through each type of element. However, they do have differences that become apparent when we begin to manipulate those elements as with CSS. This is because each element is just a node in the DOM, plain and simple, and they are treated like all other nongraphic elements: CSS doesn't affect the internals of the element,

just how it ultimately appears on the page. Individual parts of SVGs declared in markup can, in fact, be separately styled so long as they can be identified with a CSS selector. In any case, such styling only affects presentation, so if new styles are applied, they are applied to the original contents of the element.

What's also true is that graphics elements can overlap with each other and with nongraphic elements (as well as video), and the rendering engine automatically manages transparency according to the **z-index** of those elements. Each graphic element can have clear or transparent areas, as is built into image formats like PNG. In a **canvas**, any areas cleared with the **clearRect** method that isn't otherwise affected by other API calls will be transparent. Similarly, any area in an SVG's rectangle that's not affected by its individual parts will be transparent.

As an example, Scenario 2 in the example allows you to toggle a few styles (with a check box) on the same elements shown earlier. In this case, I've left the background of the canvas element transparent so that we can see areas that show through. When the styles are applied, the **img** element gets is rotated and transformed, the **canvas** gets scaled, and individual parts of the **svg** are styled with new colors, as shown in Figure 10-2.

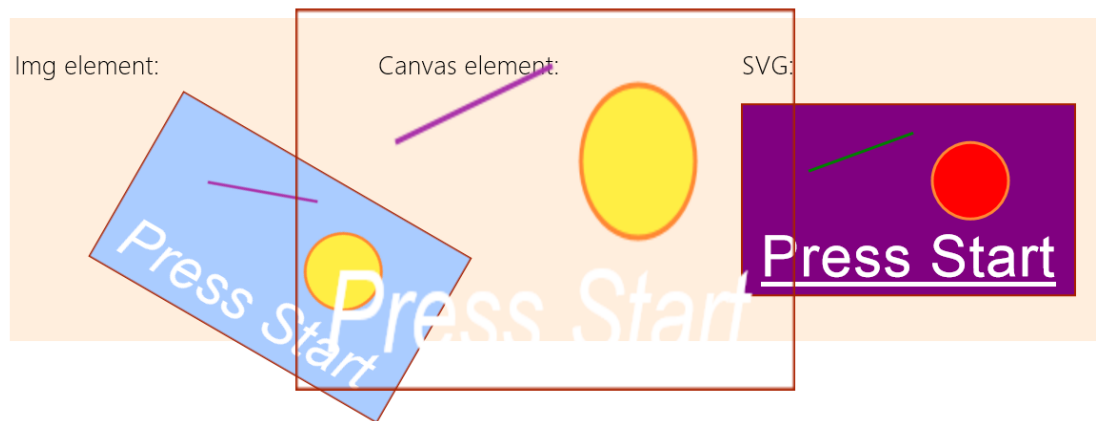


FIGURE 10-2 Styles applied to graphic elements; individual parts of the SVG can be styled if they are accessible through the DOM.

The styles in `scenario2.css` are simple:

```
.transformImage {  
  transform: rotate(30deg) translateX(120px);  
}  
  
.scaleCanvas {  
  transform: scale(1.5, 2);  
}
```

as is the code in `scenario2.js` that applies them:

```
function toggleStyles() {
```

```

var applyStyles = document.getElementById("check1").checked;

document.getElementById("image1").className = applyStyles ? "transformImage" : "";
document.getElementById("canvas1").className = applyStyles ? "scaleCanvas" : "";

document.getElementById("r").style.fill = applyStyles ? "purple" : "";
document.getElementById("l").style.stroke = applyStyles ? "green" : "";
document.getElementById("c").style.fill = applyStyles ? "red" : "";
document.getElementById("t").style.fontStyle = applyStyles ? "normal" : "";
document.getElementById("t").style.textDecoration = applyStyles ? "underline" : "";
}

```

The other thing you might have noticed when the styles are applied is that the scaled-up canvas looks rasterized, like a bitmap would typically be. This is expected behavior, as shown in the following table of scaling characteristics, which are demonstrated in Scenarios 3 and 4 of the example.

Element	Scaling	Handling layout changes for best appearance
<code>img</code>	rasterized	Change <code>src</code> attribute for different scales (or just use an SVG file as a source).
<code>canvas</code>	rasterized	Redraw canvas using scaled dimensions; this is often best done by calling <code><context>.scale</code> according to the needed display size while using the same coordinates in the rest of the code.
<code>svg</code>	smooth	Not needed. Use <code>viewBox</code> and <code>preseveAspectRatio</code> for proportional scaling.

Additional Characteristics of Graphics Elements

There are a few additional characteristics to be aware of with graphics elements. First, different kinds of operations will trigger a re-rendering of the element in the document. Second is the mode of operation of each element. Third are the relative strengths of each element. These are summarized in the following table:

Element	Trigger for re-rendering*	Mode	Strengths
<code>img</code>	Change <code>src</code> attribute Change of styling via JavaScript	Pixel	Fast to render and transform Great for static elements and static/repeating backgrounds Sprite animation by changing <code>src</code> attribute
<code>canvas</code>	Calls to context API Change of styling via JavaScript Note: re-rendering only happens when code returns control to the host and unblocks the UI thread; there are no visible changes while the code is manipulating the canvas.	Immediate: API calls are rendered to pixels and forgotten	Fine-grained dynamic content Fast to render after being drawn Pixel-level manipulation Excellent for fine-grained dynamic/interactive content with frequent computation
<code>svg</code>	Change to element structure Change of styling via JavaScript	Retained: all shapes exist as DOM elements (unless used as <code>img src</code>)	Smooth scaling Fine-grained control over individual (retained) elements Shape-level manipulation Excellent for interactive graphics, detailed and scalable

Sidebar: Using Media Queries to Show and Hide SVG Elements

Because SVGs generate elements in the DOM, those elements can be individually styled. You can use this fact along with media queries to hide different parts of the SVG depending on its size. To do this, add different classes to those SVG elements. Then, in CSS, add or remove the `display: none` style for those classes within media queries like `@media (min-width:300px)` and `(max-width:499px)`. You may need to do some calculating for the size of the SVG relative to the app window, but it means that you can effectively remove detail from an SVG rather than allowing those parts to be rendered with just a few pixels.

In the end, the reason why HTML5 has all three of these elements is because all three are really needed. All of them also benefit from full hardware acceleration, just as they do in Internet Explorer, since WinRT apps written in HTML and JavaScript run on the same rendering engine as the browser.

The best practice in app design is to really explore the appropriate use of each type of elements. Each element can have transparent areas, so you can easily achieve some very fun effects. For example, if you have data that maps video timings to caption or other text, you can simply use an interval handler (with the interval set to the necessary granularity like a half-second) to take the video's `currentTime` property, retrieve the appropriate text for that segment, and render the text to an otherwise transparent canvas that sits on top of the video. Titles and credits can be done in a similar manner.

Some Tips and Tricks

Working with the HTML graphics elements is generally straightforward, but knowing some details can help you in working with them inside a WinRT app.

Img Elements

- Use the `title` attribute of `img` for tooltips, not the `alt` attribute. You can also use a `WinJS.UI.Tooltip` control as described in Chapter 4, “Controls, Control Styling, and Data Binding.”
- To create an image from an in-memory stream, see [MSApp.createBlobFrom-RandomAccessStream](#), the result of which can be then given to `URL.createObjectURL` to create an appropriate URL for a `src` attribute. We'll encounter this elsewhere in this chapter, and we'll need it when working with the Share contract in Chapter 12, “Contracts.” The same technique also works for audio and video streams.
- When loading images from `http://` or other remote sources, you run the risk of having the element show a red X placeholder image. To prevent this, catch the `img.onerror`

event and supply your own placeholder:

```
var myImage = document.getElementById('image');
myImage.onerror = function () { onImageError(this);}

function onImageError(source) {
    source.src = "placeholder.png";
    source.onerror = "";
}
```

Svg Elements

- `<script>` tags are not supported within `<svg>`.
- If you have an SVG file, you can load it into an `img` element by pointing at the file with the `src` attribute, but this doesn't let you traverse the SVG in the DOM. If you want the latter behavior, load the SVG in an `iframe` instead. The SVG contents will then be within that element's `contentDocument.documentElement` property:

```
<!-- in HTML-->
<iframe id="Mysvg" src="myFolder/mySVGFile.svg" />

// in JavaScript
var svg = document.getElementById("Mysvg").contentDocument.documentElement;
```

- PNGs generally perform better than SVGs, so if you don't technically need an SVG or have a high-performance scenario, consider using scaled PNGs. Or you can dynamically create a static image (appropriately scaled to the current resolution) from an SVG so as to use the image for faster rendering later:

```
<!-- in HTML-->

<canvas id="canvas" style="display: none;" />

// in JavaScript
var c = document.getElementById("canvas").getContext("2d");
c.drawImage(document.getElementById("svg"),0,0);
var imageURLToUse = document.getElementById("canvas").toDataURL();
```

- Two helpful SVG references (JavaScript examples): <http://www.carto.net/papers/svg/samples/> and <http://srufaculty.sru.edu/david.dailey/svg/>.

Canvas Elements

All the function names mentioned here are methods of a canvas's context object:

- Remember that a `canvas` element needs specific `width` and `height` attributes (in JavaScript, `canvas.width` and `canvas.height`), not dimension styles. It does not accept px, em, %, or other units.
- Despite its name, the `closePath` method is *not* a direct complement to `beginPath`.

`beginPath` is used to start a new path that can be stroked, clearing any previous path. `closePath`, on the other hand, simply connects the two endpoints of the current path, as if you did a `lineTo` between those points. It does *not* clear the path or start a new one. This seems to confuse programmers quite often, which is why you sometimes see a circle drawn with a line to the center!

- Paths must be stroked with a call to `stroke` in order to render; until that time, think of them as a pencil sketch of something that's not been inked in. Note also that stroking implies a call to `beginPath`.
- When animating on canvas, doing `clearRect` on the entire canvas and redrawing every frame is generally easier to work with than clearing many small areas and redrawing individual parts of the canvas. The app host eventually has to render the entire canvas in its entirety with every frame anyway to manage transparency, so trying to optimize performance by clearing small rectangles isn't an effective strategy except when you're only doing a small number of API calls for each frame.
- Rendering canvas API calls is accomplished by converting them to the equivalent Direct2D calls in the GPU. This draws shapes with automatic antialiasing. As a result, drawing a shape like a circle in a color and drawing the same circle with the background color does *not* erase every pixel. To effectively erase a shape, use `clearRect` on an area that's slightly larger than the shape itself. This is one reason why clearing the entire canvas and redrawing every frame often ends up being easier.
- To set a background image in a canvas (that you don't have to draw each time), you can use the `canvas.style.backgroundImage` property with an appropriate URL to the image.
- When using `drawImage`, you may need to wait for the source image to load using code such as this:

```
var img = new Image();
img.onload = function () { myContext.drawImage(myImg, 0, 0); }
myImg.src = "myImageFile.png";
```

- Although other graphics APIs see a circle as a special case of an ellipse (with x and y radii being the same), the canvas `arc` function works with circles only. Fortunately, a little use of scaling makes it easy to draw ellipses, as shown in the utility function below. Note that we use `save` and `restore` so that the `scale` call *only* applies to the `arc`; it does not affect the `stroke` that's used from `main`. This is important, because if the scaling factors are still in effect when you call `stroke`, the line width will vary instead of remaining constant.

```
function arcEllipse(ctx, x, y, radiusX, radiusY, startAngle, endAngle, anticlockwise) {
  //Use the smaller radius as the basis and stretch the other
  var radius = Math.min(radiusX, radiusY);
  var scaleX = radiusX / radius;
```



```

    var scaleY = radiusY / radius;

    ctx.save();
    ctx.scale(scaleX, scaleY);

    //Note that centerpoint must take the scale into account
    ctx.arc(x / scaleX, y / scaleY, radius, startAngle, endAngle, anticlockwise);
    ctx.restore();
}

```

- By copying pixel data from a video, it's possible with the canvas to dynamically manipulate a video (without affecting the source, of course). This is processor-intensive, but it's a useful technique.

Here's an example of frame-by-frame video manipulation, the technique for which is nicely outlined in a Windows team blog post, [Canvas Direct Pixel Manipulation](#).⁵³ In the VideoEdit example for this chapter, default.html contains a `video` and `canvas` element in its main body:

```

<video id="video1" src="ModelRocket1.mp4" muted style="display: none"></video>
<canvas id="canvas1" width="640" height="480"></canvas>

```

In code (default.js), we call `startVideo` from within the activated handler. This function starts the video and uses `requestAnimationFrame` to do the pixel manipulation for every video frame:

```

var video1, canvas1, ctx;
var colorOffset = { red: 0, green: 1, blue: 2, alpha: 3 };

function startVideo() {
    video1 = document.getElementById("video1");
    canvas1 = document.getElementById("canvas1");
    ctx = canvas1.getContext("2d");

    video1.play();
    requestAnimationFrame(renderVideo);
}

function renderVideo() {
    //Copy a frame from the video to the canvas
    ctx.drawImage(video1, 0, 0, canvas1.width, canvas1.height);

    //Retrieve that frame as pixel data
    var imgData = ctx.getImageData(0, 0, canvas1.width, canvas1.height);
    var pixels = imgData.data;

    //Loop through the pixels, manipulate as needed
    var r, g, b;

    for (var i = 0; i < pixels.length; i += 4) {
        r = pixels[i + colorOffset.red];
        g = pixels[i + colorOffset.green];

```

⁵³ See also <http://beej.us/blog/2010/02/html5s-canvas-part-ii-pixel-manipulation/>.

```

    b = pixels[i + colorOffset.blue];

    //This creates a negative image
    pixels[i + colorOffset.red] = 255 - r;
    pixels[i + colorOffset.green] = 255 - g;
    pixels[i + colorOffset.blue] = 255 - b;
}

//Copy the manipulated pixels to the canvas
ctx.putImageData(imgData, 0, 0);

//Request the next frame
requestAnimationFrame(renderVideo);
}

```

Here the page contains a hidden video element (`style="display: none"`) that is told to start playing once the document is loaded (`video1.play()`). In a ~60 frames per second timer loop (using `requestAnimationFrame`), the current frame of the video is copied to the canvas (`drawImage`) and the pixels for the frame are copied (`getImageData`) into the `imgData` buffer. We then go through that buffer and negate the color values, thereby producing a photographically negative image (an alternate formula to change to grayscale is also shown in the code comments, omitted above). We then copy those pixels back to the canvas (`putImageData`) so that when we return, those negated pixels are rendered to the display.

Again, this is processor-intensive as it's not generally a GPU-accelerated process; it's much better to write a video effect DLL where possible as discussed in "Applying a Video Effect" later on. Nevertheless, it is a useful technique to know. What's really happening is that instead of drawing each frame with API calls, we're simply using the video as a data source. So we could, if we like, embellish the canvas in any other way we want before returning from the `renderVideo` function. An example of this that I really enjoy is shown in [Manipulating video using canvas](#) on Mozilla's developer site, which dynamically sets green-screen background pixels to be transparent so that an `img` element placed underneath the video shows through as a background. The same could even be used to layer two videos so that a background video is used instead of a static image.

Video Playback and Deferred Loading

Let's now talk a little more about video playback itself. As we've already seen, simply including a `video` element in your HTML, or creating an element on the fly gives you playback ability. In the code below, the video is sourced from a local file, starts playing by itself, loops continually, and provides controls:

```
<video src="media/ModelRocket1.mp4" controls loop autoplay></video>
```

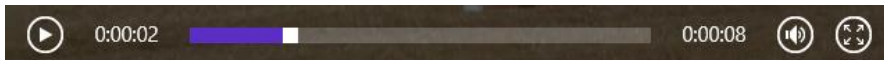
As we've been doing in this book, we're not going to rehash the details that are available in the W3C spec for the `video` and `audio` tags, starting at <http://www.w3.org/TR/html5/video.html>. This spec will give you all the properties, methods, and events for these elements; especially note the event summary in [section 4.8.10.15](#), and that most of the properties and methods for both are found in

[Media elements section 4.8.10](#). Note that the `track` element is supported for both `video` and `audio`; you can find an example of using it in Scenario 4 (demonstrating subtitles) of the [HTML Media Playback sample](#) as we won't be covering it more here.

It's also helpful to understand that `video` and `audio` are closely related, since they're part of the same spec. In fact, if you want to just play the audio portion of a video, you can use the `Audio` object in JavaScript:

```
//Play just the audio of a video
var movieAudio = new Audio("http://www.kraigbrockschmidt.com/downloads/media/ModelRocket1.mp4");
movieAudio.load();
movieAudio.play();
```

For any given video element, you can set the width and height to control the playback size (as to 100% for full-screen). This is important when your app switches between view states, and you'll likely have CSS styles for video elements in your various media queries. Also, if you have a control to play full screen, simply make the video the size of the viewport (after also calling [Windows.UI.ViewManagement.ApplicationView.tryUnsnap](#) if you're in the snapped view). In addition, when you create a video element with the `controls` attribute, it will automatically have a full-screen control on the far right that does exactly what you expect within a WinRT app:



In short, you don't need to do anything special to make this work. When the video is full screen, a similar button (or the ESC key) returns to the normal app view.

Note In case you're wondering, the audio and video elements don't provide any CSS pseudo-selectors for styling the controls bar. As my son's preschool teacher would say (in reference to handing out popsicles, but it works here), "You get what you get and you don't throw a fit and you're happy with it." If you'd like to do something different with these controls, you'll need to turn off the defaults and provide controls of your own that would call the element methods appropriately.

When implementing your own controls, be sure to set a timeout to make the controls disappear (either hiding them or changing the z-index) when they're not being used. This is especially important if you have a full-screen button for video like the built-in controls, where you would basically resize the element to match the screen dimensions. When you do this, Windows will automatically detect this full-screen video state and do some performance optimizations, but not if any other element is front of the video. It's also a good idea to disable any animations you might be running and disable unnecessary background tasks.

You can use the various events of the `video` element to know when the video is played and paused, among other things (though there is not an event for going full-screen), but you should also respond appropriately when hardware buttons for media control are being used. For this purpose, listen for events coming from the [Windows.Media.MediaControl](#) object, such as `playpressed`, `pausepressed`, and so on. Refer to the [Configure keys for media sample](#) for a demonstration, but adding the listeners generally looks like this:

```
mediaControl = Windows.Media.MediaControl;

mediaControl.addEventListener("soundlevelchanged", soundLevelChanged, false);
mediaControl.addEventListener("playpausetoggelpressed", playpause, false);
mediaControl.addEventListener("playpressed", play, false);
mediaControl.addEventListener("stoppressed", stop, false);
mediaControl.addEventListener("pausepressed", pause, false);
```

I also mentioned that you might want to defer loading a video until it's needed and show a preview image in its place. This is accomplished with the `poster` attribute, whose value is the image to use:

```
<video id="video1" poster="media/rocket.png" width="640" height="480"></video>

var video1 = document.getElementById("video1");

var clickListener = video1.addEventListener("click", function () {
    video1.src = "http://www.kraigbrockschmidt.com/downloads/media/ModelRocket1.mp4";
    video1.load();

    //Remove listener to prevent interference with video controls
    video1.removeEventListener("click", clickListener);

    video1.addEventListener("click", function () {
        video1.controls = true;
        video1.play();
    });
});
```

In this case I'm not using `preload="true"` or even providing a `src` value so that nothing is transferred until the video is tapped. When a tap occurs, that listener is removed, the video's own controls are turned on and playback is started. This, of course, is a more roundabout method; often you'll use `preload="true" controls src="..."` directly in the video element, as the `poster` attribute will handle the preview image.

Disabling Screen Savers and the Lock Screen During Playback

When playing video, especially full-screen, it's important to disable any automatic timeouts that would blank the display or lock the device. This is done through the `Windows.System.Display.DisplayRequest` object. Before starting playback, create an instance of this object and call its `requestActive` method.

```
var displayRequest = new Windows.System.Display.DisplayRequest();
displayRequest.requestActive();
```

If this call succeeds, you'll be guaranteed that the screen will stay active despite user interactivity. When the video is complete, be sure to call `requestRelease`. Note that Windows will automatically deactivate such requests when your app is moved to the background, and it will reactivate them when the user switches back.

Video Element Extension APIs

Beyond the HTML5 standards for [video](#) elements, some additional properties and methods are added to them in Windows 8, as shown in the following table and documented on the [video element](#) page. Also note the references to the [HTML Media Playback sample](#) where you can find some examples of using these.

Properties	Description
msHorizontalMirror	A Boolean that controls whether the playback is flipped horizontally. This is particularly useful when sourcing the video element from a camera to make sure the user sees the proper orientation.
msZoom	A Boolean that indicates whether to allow the video element to fit inside its display space by trimming the top/bottom or left/right (when true). This allows apps to give users control over videos whose aspect ratio differs from that of its given display area—that is, to remove letterboxing or sidepillars. For a demonstration, refer to Scenario 3 of the HTML Media Playback sample .
msIsLayoutOptimalForPlayback (onMSVideoOptimalLayoutChanged)	A Boolean that indicates whether a video will have the best playback based on its layout. When this changes the onMSVideoOptimalLayoutChanged event fires. For details, see How to optimize video rendering and Audio and Video Performance .
msIsStereo3D	A Boolean that indicates whether the system considers the video element's source to be 3D (based on metadata in the video itself). Whether the system itself is capable can be determined through Windows.Graphics.Display.DisplayProperties.stereoEnabled . Apps can also listen for Windows.Graphics.Display.DisplayProperties.stereoEnabledChanged to know when the capabilities change. For details on this and other Stereo 3D concerns, refer to How to enable stereo video playback and Scenario 5 of the HTML Media Playback sample .
msStereo3DRenderMode	Can be mono (the default) or stereo so that apps can control playback. (See above for references.)
msStereo3DPackingMode	Can be none (2D default), topbottom , or sidebyside ; this is an adjustment available to apps when the video metadata doesn't clearly indicate which orientation to use. (See above for references.)
msRealtime	Enables the media to reduce initial latency as much as possible for playback. This is important for two-way communication apps, for example as well as gaming chat, but should be used carefully. For details, refer to How to enable low-latency playback and the Real-time Communications sample .
msPlayTo msPlayToDisabled msPlayToPrimary msPlayToSource	Properties related to Windows' PlayTo feature. See the "PlayTo" section at the end of this chapter. Note that these are also available on img and audio elements as well.
msAudioTracks	An array of audio track descriptions to support additional languages or other tracks (e.g., commentary). Set msAudioTracks.selectedTrack to the desired index to change the playback audio. For details, refer to How to select audio tracks in different languages as well as Scenario 2 of the HTML Media Playback sample .
msAudioCategory	Identifies the kind of audio being played in the video; see "Playback Manager and Background Audio" later for the specific values. Note that setting this to "Communications" will also set the device type to "Communications" and force msRealtime to true .
msAudioDeviceType	Specifies the output devices that audio will be sent to; see "Audio Element Extension APIs."

Methods	Description
<code>msFrameStep</code> (<code>onMSVideoFrameStepCompleted</code>)	Steps the video by one frame forward or backward. The <code>onMSVideoFrameStepCompleted</code> event fires when the step is complete.
<code>msInsertVideoEffect</code> <code>msInsertAudioEffect</code> <code>msClearEffects</code>	Adds or removes effects during playback (see below). All are available on <code>video</code> ; <code>msInsertVideoEffect</code> is not available on <code>audio</code> elements.
<code>msSetMediaProtectionManager</code>	Used for DRM with both <code>audio</code> and <code>video</code> ; see “Streaming from a Server and Digital Rights Management (DRM)” toward the end of this chapter.
<code>msSetVideoRectangle</code>	Sets the dimension of a subrectangle within a video.
<code>onMSVideoFrameStepCompleted</code> (event)	Occurs when the video format changes.

The Source Attribute and Custom Codecs

Video (and audio) elements can use the HTML5 `source` attribute. In web applications, multiple source elements are used to provide alternate video formats in case a client system doesn’t have the necessary codec for the primary source. Given that the list of supported formats in Windows is well known (refer again to [Supported audio and video formats](#)), this isn’t much of a concern for WinRT apps. However, source is still useful because it can identify the specific codecs for the source:

```
<video controls loop autoplay>
  <source src="video1.vp8" type="video/webm" />
</video>
```

This is important when you need to provide a custom codec for your app through `Windows.Media.MediaExtensionManager`, outlined in the “Custom Decoders/Encoders and Scheme Handlers” section later in this chapter, as the codec identifies the extension to load for decoding. I show WebM as an example here because it’s not directly available to WinRT apps (though it is in Internet Explorer). When the app host running a WinRT app encounters the `video` element above, it will look for a matching decoder for the specified `type`.

Applying a Video Effect

The earlier table shows that video elements have `msInsertVideoEffect` and `msInsertAudioEffect` methods on them. WinRT provides a built-in video stabilization effect that is easily applied to an element, as demonstrated in Scenario 3 of the [Media extensions sample](#) (which plays the same video with and without the effect, so the stabilized one is muted):

```
vidStab.msClearEffects();
vidStab.muted = true;
vidStab.msInsertVideoEffect(Windows.Media.VideoEffects.videoStabilization, true, null);
```

Custom effects, as demonstrated in Scenario 4 of the sample, are implemented as separate *dynamic-link libraries* (or *DLLs*), typically in C++ for best performance, and are included in the app

package because a WinRT app can install a DLL only for its own use and not for systemwide access. With the sample you'll find DLL projects for a grayscale, invert, and geometric effects, where the latter has three options for fisheye, pinch, and warp. In the CustomEffect.js file you can see how these are applied, with the first parameter to `msInsertVideoEffect` being a string that identifies the effect as exported by the DLL (see, for instance, the `InvertTransform.idl` file in the `InvertTransform` project):

```
vid.msInsertVideoEffect("GrayscaleTransform.GrayscaleEffect", true, null);

vid.msInsertVideoEffect("InvertTransform.InvertEffect", true, null);
```

The second parameter to `msInsertVideoEffect`, by the way, indicates whether the effect is required, so it's typically `true`. The third is a parameter called *config*, which just contains additional information to pass to the effect. In the case of the geometric effects in the sample, this parameter specifies the particular variation:

```
var effect = new Windows.Foundation.Collections.PropertySet();
effect["effect"] = effectName;
vid.msClearEffects();
vid.msInsertVideoEffect("PolarTransform.PolarEffect", true, effect);
```

where `effectName` will be either "Fisheye", "Pinch", or "Warp".

Audio effects, not shown in the sample, are applied the same way with `msInsertAudioEffect` (with the same parameters). Do note that each element can have at most two effects per media stream. A `video` element can have two video effects and two audio effects; an `audio` element can have two audio effects. If you try to add more, the methods will throw an exception. This is why it's a good idea to call `msClearEffects` before inserting any others.

For additional discussion on effects and other media extensions, see [Using media extensions](#).

Browsing Media Servers

Many households, including my own, have one or more media servers available on the local network from which apps can play media. Getting to these servers is the purpose of the one other property in `Windows.Storage.KnownFolders` that we haven't mentioned yet: `mediaServerDevices`. As with other known folders, this is simply a `StorageFolder` object through which you can then enumerate or query additional folders and files. In this case, if you call its `getFoldersAsync`, you'll receive back a list of available servers, each of which is represented by another `StorageFolder`. From there you can use file queries, as discussed in Chapter 8, "State, Settings, Libraries, and Documents," to search for the types of media you're interested in or apply user-provided search criteria.

An example of this can be found in the [Media Server client sample](#) of the Windows SDK.

Audio Playback and Mixing

As with video, the `audio` element provides its own playback abilities, including controls, looping, and

autoplay:

```
<audio src="media/SpringyBoing.mp3" controls loop autoplay></audio>
```

Again, as described earlier, the same W3C spec applies to both `video` and `audio` elements. The same code to play just the audio portion of a video is exactly what we use to play an audio file:

```
var sound = new Audio("media/SpringyBoing.mp3");
sound.msAudioCategory = "SoundEffect";
sound.load(); //For pre-loading media
sound.play(); //At any later time
```

As also mentioned before, creating an `Audio` object without controls and playing it has no effect on layout, so this is what's generally used for sound effects in games and other apps.

As with video, it's important for apps that do audio playback to respond appropriately to the events coming from the [Windows.Media.MediaControl](#) object, especially `playpressed`, `pausepressed`, `stoppressed`, and `playpausetogglepressed`. This lets the user control audio playback with hardware buttons, which you would use when playing music tracks, for instance. However, you would not apply these events to other kinds of audio, such as game sounds.

Speaking of which, an interesting aspect of audio is how to mix multiple sounds together, as games generally require. Here it's important to understand that each `audio` element can be playing one sound: it only has one source file and one source file alone. However, multiple `audio` elements can be playing at the same time with automatic intermixing depending on their assigned categories. (See "Playback Manager and Background Audio" below.) In this example, some background music plays continually (`loop` is set to true, and the volume is halved) while another sound is played in response to taps (see the `AudioPlayback` example with this chapter's content):⁵⁴

```
var sound1 = new Audio("media/SpringyBoing.mp3");
sound1.load(); //For pre-loading media

//Background music
var sound2 = new Audio();
sound2.msAudioCategory = "ForegroundOnlyMedia"; //Set this before setting src
sound2.src = "http://www.kraigbrockschmidt.com/mp3/WhoIsSylvia_Portland0R_5-06.mp3";
sound2.loop = true;
sound2.volume = 0.5; //50%;
sound2.play();

document.getElementById("btnSound").addEventListener("click", function () {
    //Reset position in case we're already playing
    sound1.currentTime = 0;
    sound1.play();
});
```

By loading the tap sound when the object is created, we know we can play it at any time. When

⁵⁴ And yes, I am playing the guitar and singing the lead part in this live recording, along with my friend Ted Cutler. The song, *Who is Sylvia?*, was composed by another friend, J. Donald Walters, using lyrics of Shakespeare.

initiating playback, it's a good idea to set the `currentTime` to 0 so that the sound always plays from the beginning.

The question with mixing, especially in games, really becomes how to manage many different sounds without knowing ahead of time how they will be combined. You may need, for instance, to overlap three playbacks of the same sound with different starting times, but it's impractical to declare three audio elements with the same source.

The technique that's emerged is to use "rotating channels" [as described on the Ajaxian website](#). To summarize:

1. Declare `audio` elements for each *sound* (with `preload="auto"`).
2. Create a pool (array) of `Audio` objects for however many simultaneous channels you need.
3. To play a sound:
 - a. Obtain an available `Audio` object from the pool.
 - b. Set its `src` attribute to one that matches a preloaded `audio` elements.
 - c. Call that pool object's `play` method.

As sound designers in the movies have discovered, it is possible to have too much sound going on at the same time, because it gets really muddled. So you may not need more than a couple dozen channels at most.

Hint Need some sounds for your app? Check out <http://www.freesound.org>.

Audio Element Extension APIs

As with the `video` element, a few extensions are available on `audio` elements as well, namely those to do with effects (`msInsertAudioEffect`), DRM (`msSetMediaProtectionManager`), PlayTo (`msPlayToSource`, etc.), `msRealtime`, and `msAudioTracks`, as listed earlier in "Video Element Extension APIs." In fact, every extension API for `audio` exists on `video`, but two of them have primary importance for `audio`:

- `msAudioDeviceType` Allows an app to determine which output device audio will render to: `"Console"` (the default) and `"Communications"`. This way an app that knows it's doing communication (like chat) doesn't interfere with media audio.
- `msAudioCategory` Identifies the type of audio being played (see table in the next section). This is very important for identifying whether audio should continue to play in the background (thereby preventing the app from being suspended), as described in the next section. Note that you should *always* set this property before setting the audio's `src` and that setting this to `"Communications"` will also set the device type to `"Communications"` and force `msRealtime` to `true`.

Do note that despite the similarities between the values in these properties, `msAudioDeviceType` is for selecting an output device whereas `msAudioCategory` identifies the *type* of audio that's being played through whatever device. A communications category audio could be playing through the console device, for instance, or a media category could be playing through the communications device. The two are separate concepts.

Playback Manager and Background Audio

To explore different kinds of audio playback, let's turn our attention to the [Playback Manager msAudioCategory sample](#) in the Windows SDK. I won't show a screen shot of this because, doing nothing but audio, there isn't much to show! Instead, let me outline what its different scenarios demonstrate in terms of audio behavior in the following table, as well as list those categories that aren't represented in the sample but that can be used in your own app. In each scenario you need to first select an audio file through the file picker.

Scenario	msAudioCategory	Description
1	<code>BackgroundCapableMedia</code>	Plays the selected audio when the app is both visible and in the background. With this category, the app will not be suspended when in the background, which you can confirm through Task Manager. This is typically used for playing local playlists, local or streaming media files, music videos, etc.
2	<code>Communications</code>	Like <code>BackgroundCapableMedia</code> , this will also continue to play the selected audio when the app is in the background. Use this for peer-to-peer chat, VoIP, etc.
3	<code>Other</code> (the default for <code>audio</code> elements)	Plays the selected audio when the app is in the foreground, mixing with background audio; the audio is paused when the app is in the background.
4	<code>ForegroundOnlyMedia</code>	Plays the selected audio when the app is in the foreground; the audio is paused when the app is in the background. When audio of this category is played, background audio will be muted.
5	<code>Alert</code>	Plays the selected audio when the app is in the foreground and attenuates background audio. This is used for app notifications like ringtones as well as system alerts.
n/a	<code>GameMedia</code>	Used for background music in a game.
n/a	<code>GameEffects</code>	Used for game sound effects intended to mix with existing audio (all nonmusic sounds).
n/a	<code>SoundEffects</code>	Other sound effects (outside of games) intended to mix in with existing audio, such as brief dings, beeps, boinks, and blurps that indicate activity but don't otherwise quality as alerts.

Where a single audio stream is concerned, there isn't always a lot of difference between some of these categories. Yet as the table indicates, different categories have different effects on other simultaneous audio streams. For the purpose, the Windows SDK does an odd thing by providing a second identical sample to the first, [Playback Manager msAudioCategory sample\(2\)](#). This allows you

run these apps at the same time (one in snapped view, the other in filled view) and play audio with different category settings to see how they combine.

How different audio streams combine is a subject that's discussed in the [Audio Playback in a Windows 8 App whitepaper](#). However, what's most important is that you assign the most appropriate category to any particular audio stream. These categories help the playback manager perform the right level of mixing between audio streams according to user expectations, both with multiple streams in the same app, and streams coming from multiple apps. For example, users will expect that alarms, being an important form of notification, will temporarily attenuate other audio streams. Similarly, users will expect that an audio stream of a foreground app will take precedence over a stream of the same category of audio playing in the background. As a developer, then, avoid playing games with the categories. Just assign the most appropriate category to your audio stream so that the playback manager can do its job with audio from all sources and deliver a consistent experience for the entire system.

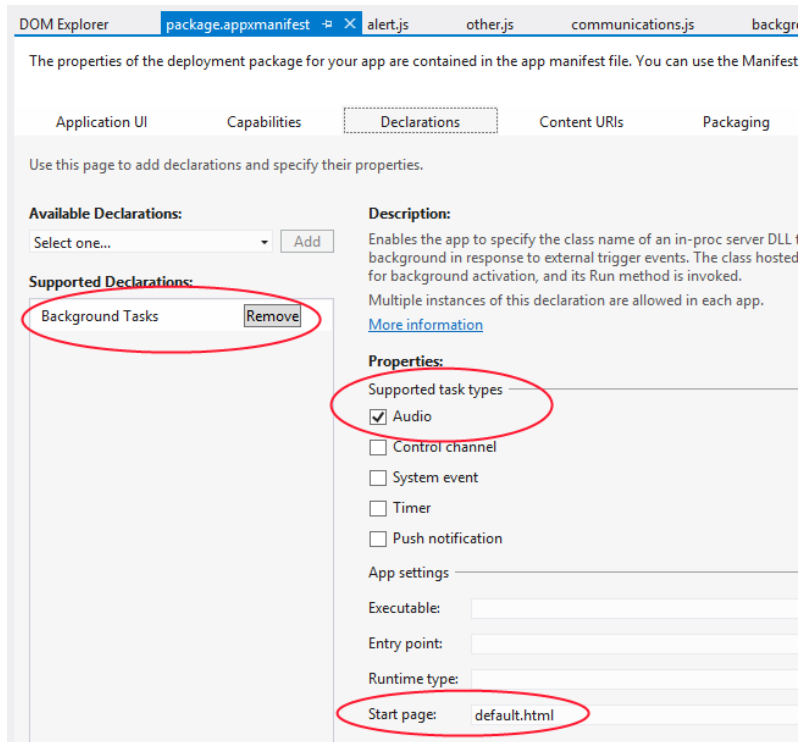
Setting an audio category for any given `audio` element is a simple matter of setting its `msAudioCategory` attribute. Every scenario in the samples does the same thing for this, making sure to set the category before setting the `src` attribute (shown here from `backgroundcapablemedia.js`):

```
audtag = document.createElement('audio');
audtag.setAttribute("msAudioCategory", "BackgroundCapableMedia");
audtag.setAttribute("src", fileLocation);
```

You could accomplish the same thing in markup, of course. Some examples:

```
<audio id="audio1" src="song.mp3" msAudioCategory="BackgroundCapableMedia"></audio>
<audio id="audio2" src="voip.mp3" msAudioCategory="Communications"></audio>
<audio id="audio3" src="lecture.mp3" msAudioCategory="Other"></audio>
```

With `BackgroundCapableMedia` and `Communications`, however, simply setting the category isn't sufficient: you also need to declare an audio background task extension in your manifest. This is easily accomplished by going to the Declarations tab in the manifest designer:



First, select Background Tasks from the Available Declarations drop-down list. Then check Audio under Supported Task Types, and identify a Start page under App Settings. This latter page isn't really essential for background audio (because you'll never be launched for this purpose), but you need to provide something to make the manifest editor happy.

These declarations appear as follows in the manifest XML, should you care to look:

```
<Application Id="App" StartPage="default.html">
  <!-- ... -->
  <Extensions>
    <Extension Category="windows.backgroundTasks" StartPage="default.html">
      <BackgroundTasks>
        <Task Type="audio" />
      </BackgroundTasks>
    </Extension>
  </Extensions>
</Application>
```

Furthermore, background audio *must* also add listeners for the [Windows.Media.MediaControl](#) events that we've already mentioned so that the app can properly control playback when asked. They're required because they make it possible for the playback manager to control the audio streams as the user switches between apps. Otherwise, your audio will always be paused and muted when the app goes into the background.

How to do this is shown in the Playback Manager sample for all its scenarios; the following is from communications.js (some code omitted):

```
mediaControl = Windows.Media.MediaControl;

mediaControl.addEventListener("soundlevelchanged", soundLevelChanged, false);
mediaControl.addEventListener("playpausetogglepressed", playpause, false);
mediaControl.addEventListener("playpressed", play, false);
mediaControl.addEventListener("stoppressed", stop, false);
mediaControl.addEventListener("pausepressed", pause, false);

// audTag variable is the global audio element for the page

function playpause() {
    if (!audtag.paused) {
        audtag.pause();
    } else {
        audtag.play();
    }
}

function play() {
    audtag.play();
}

function stop() {
    // Nothing to do here
}

function pause() {
    audtag.pause();
}

function soundLevelChanged() {
    //Catch SoundLevel notifications and determine SoundLevel state.
    //If it's muted, we'll pause the player.
    var soundLevel = Windows.Media.MediaControl.soundLevel;

    //No actions are shown here, but the options are spelled out to show the enumeration
    switch (soundLevel) {
        case Windows.Media.SoundLevel.muted:
            break;
        case Windows.Media.SoundLevel.low:
            break;
        case Windows.Media.SoundLevel.full:
            break;
    }

    appMuted();
}

function appMuted() {
    if (audtag) {
```

```

        if (!audtag.paused) {
            audtag.pause();
        }
    }
}

```

Technically speaking, a handler for `soundlevelchanged` is not required here, but the other four are. Such a minimum implementation is part of the AudioPlayback example with this chapter.

Playing Sequential Audio

An app that's playing audio tracks (such as music, an audio book, or recorded lectures) will often have a list of tracks to play sequentially, including while the app is running in the background. In this case it's important to start the next track quickly because Windows will otherwise suspend in 10 seconds after the current audio is finished. For this purpose, listen for the `audio` element's `ended` event and set the `audio.src` attribute to the next track. A good optimization in this case is to create a second Audio object and set its `src` attribute after the first track starts to play. This way that second track will be preloaded and ready to go right away, thereby avoiding potential delays in playback between tracks. This is shown in the AudioPlayback example for this chapter, where I've split the one complete song into four segments for continuous playback:

```

function playSegments() {
    var playlist = ["media/segment1.mp3", "media/segment2.mp3", "media/segment3.mp3", "media/segment4.mp3"];
    var curSong = 0;

    //Pause the other music
    document.getElementById("musicPlayback").pause();

    //Set up the first track
    var audio1 = document.getElementById("audioSegments");
    setMediaControl(audio1);
    audio1.src = playlist[0];

    //Show the element and play (it's initially hidden)
    audio1.style.display = "";
    audio1.volume = 0.5; //50%;
    audio1.play();

    //Preload the next track in readiness for the switch
    var preload = document.createElement("audio");
    preload.setAttribute("preload", "auto");
    preload.src = playlist[1];

    //Switch to the next track as soon as one had ended.
    audio1.addEventListener("ended", function () {
        curSong++;

        if (curSong < playlist.length) {
            //This track should already be loaded
            audio1.src = playlist[curSong];
            audio1.play();
        }
    });
}

```

```

        //Set up the next preload
        var nextTrack = curSong + 1;

        if (nextTrack < playlist.length) {
            preload.src = playlist[nextTrack];
        }
    }
});
}

```

Note When playing sequential tracks from a WinRT app written in JavaScript and HTML, you might notice very brief gaps between the tracks, especially if the first track flows directly into the second. This is a present limitation of the platform given the layers that exist between the HTML [audio](#) element and the low-level XAudio2 APIs that are ultimately doing the real work. You can mitigate the effects to some extent—for example, you can cross-fade the two tracks or cross-fade a third overlay track that contains a little of the first and a little of the second track. You can also use a negative time offset to start playing the next track slightly before the previous one ends. But if you a truly seamless transition, you'll need to use the XAudio2 APIs from a WinRT component to set up direct playback and bypass the [audio](#) element.

Playlists

The multitrack playback example in the previous section is clearly contrived because an app wouldn't typically have an in-memory playlist. More likely an app would load an existing playlist or create one from files that a user has selected.

WinRT supports these actions through a simple API in the [Windows.Media.Playlists](#) namespace, using the WPL (Windows Media Player), ZPL (Zune), and M3U formats. The [Playlist sample](#) in the Windows SDK (which almost wins the prize for the *shortest* sample name!) shows how to perform various tasks with the API. In Scenario 1 it lets you choose multiple files by using the file picker, creates a new [Windows.Media.Playlists.Playlist](#) object, adds those files to its [files](#) list (a vector of [StorageFile](#) objects), and saves the playlist with its [saveAsAsync](#) method (this code from create.js is simplified and reformatted a bit):

```

function pickAudio() {
    var picker = new Windows.Storage.Pickers.FileOpenPicker();
    picker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.musicLibrary;
    picker.fileTypeFilter.replaceAll(SdkSample.audioExtensions);

    picker.pickMultipleFilesAsync().done(function (files) {
        if (files.size > 0) {
            SdkSample.playlist = new Windows.Media.Playlists.Playlist();

            files.forEach(function (file) {
                SdkSample.playlist.files.append(file);
            });
        }
    });
}

```

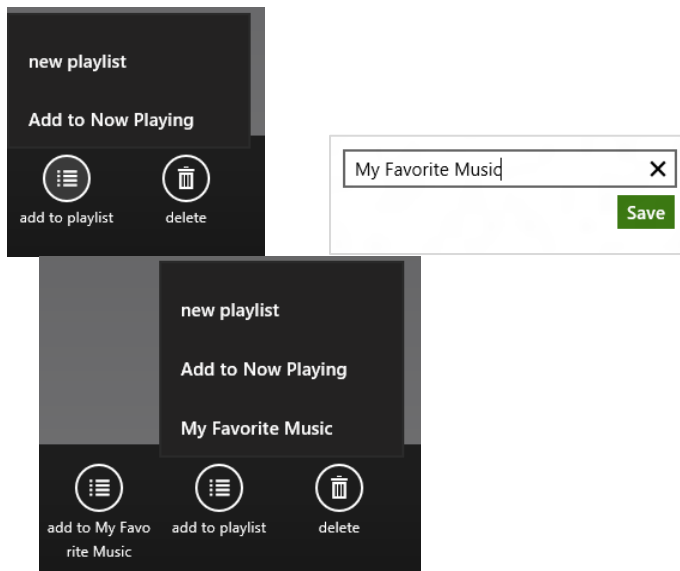
```

        SdkSample.playlist.saveAsAsync(Windows.Storage.KnownFolders.musicLibrary,
            "Sample", Windows.Storage.NameCollisionOption.replaceExisting,
            Windows.Media.Playlists.PlaylistFormat.windowsMedia)
            .done();
    }
}

```

Notice that `saveAsAsync` takes a `StorageFolder` and a name for the file (along with an optional format parameter). This accommodates a common use pattern for playlists where a music app has a single folder where it stores playlists and provides users with a simple means to name them and/or select them. In this way, playlists aren't typically managed like other user data files where one always goes through a file picker to do a Save As into an arbitrary folder. You could use `FileSavePicker`, of course, get a `StorageFile`, and then use its `path` property to get to the appropriate `StorageFolder`, but more likely you'll just save playlists in one place and present them as entities that appear only within the app itself.

For example, the Music app that comes with Windows 8 allows you create a new playlist when you're viewing tracks of some album. The following commands appear on the app bar (left), and when you select New Playlist, a flyout appears (middle) requesting the name, after which the flyout appears on the app bar (right):



The playlist then appears within the app as another album. In other words, though playlists might be saved in discrete files, they aren't necessarily presented that way to the user, and the API reflects that usage pattern.

Loading a playlist uses the `Playlist.loadAsync` method given a `StorageFile` for the playlist. This might be a `StorageFile` obtained from a file picker or from the enumeration of the app's private playlist folder. Scenario 2 of the Playlist sample (`display.js`) demonstrates the former, where it then goes

through each file and requests their music properties:

```
function displayPlaylist() {
    var picker = new Windows.Storage.Pickers.FileOpenPicker();
    picker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.musiclibrary;
    picker.fileTypeFilter.replaceAll(SdkSample.playlistExtensions);

    var promiseCount = 0;

    picker.pickSingleFileAsync()
        .then(function (item) {
            if (item) {
                return Windows.Media.Playlists.Playlist.loadAsync(item);
            }
            return WinJS.Promise.wrapError("No file picked.");
        })
        .then(function (playlist) {
            SdkSample.playlist = playlist;
            var promises = {};

            // Request music properties for each file in the playlist.
            playlist.files.forEach(function (file) {
                promises[promiseCount++] = file.properties.getMusicPropertiesAsync();
            });

            // Print the music properties for each file. Due to the asynchronous
            // nature of the call to retrieve music properties, the data may appear
            // in an order different than the one specified in the original playlist.
            // To guarantee the ordering we use Promise.join with an associative array
            // passed as a parameter, containing an index for each individual promise.
            return WinJS.Promise.join(promises);
        })
        .done(function (results) {
            var output = "Playlist content:\n\n";

            var musicProperties;
            for (var resultIndex = 0; resultIndex < promiseCount; resultIndex++) {
                musicProperties = results[resultIndex];
                output += "Title: " + musicProperties.title + "\n";
                output += "Album: " + musicProperties.album + "\n";
                output += "Artist: " + musicProperties.artist + "\n\n";
            }

            if (resultIndex === 0) {
                output += "(playlist is empty)";
            }

        }, function (error) {
            // ...
        });
}
```

We'll come back to working with these special properties in the next section, as the process also applies to other types of media.

The other method for managing a playlist is `Playlist.saveAsync`, which takes a single `StorageFile`. This is what you'd use if you've loaded and modified a playlist and simply want to save those changes (typically done automatically when the user adds or removes items from the playlist). This is demonstrated in Scenarios 3, 4, and 5 of the sample (`add.js`, `remove.js`, and `clear.js`), which just use methods of the `Playlist.files` vector like `append`, `removeAtEnd`, and `clear`, respectively.

Playback of a playlist depends, of course, on the type of media involved, but typically you'd load a playlist and sequentially take the next `StorageFile` object from its `files` vector, pass it to `URL.createObjectURL`, and then assign that URL to the `src` attribute of an `audio` or `video` element. You could also use playlists to manage lists of images for specific slide shows as well.

Loading and Manipulating Media

A user might store media files anywhere, but images, music, and videos are typically stored in the user's Pictures, Music, and Videos libraries specifically. Simply said, these are the folders that media apps should use by default until the user indicates otherwise through a folder picker. As we saw in Chapter 8, apps can declare programmatic access to the pictures, music, and videos library in its manifest and acquire the `StorageFolder` objects for these through `Windows.Storage.KnownFolders`:

```
var picsLib = Windows.Storage.KnownFolders.picturesLibrary;  
var musicLib = Windows.Storage.KnownFolders.musicLibrary;  
var vidsLib = Windows.Storage.KnownFolders.videosLibrary;
```

A photos app will typically declare the capability for the Pictures library and display those contents in a `ListView` by default. A music and videos app will do the same for their respective libraries, as you can see in the built-in Photos, Music, and Videos apps in Windows 8. Remember too that if you forget to declare the appropriate capabilities, the lines of code above will throw access denied exceptions. You'll know right away if you forgot these important details.

I should warn you ahead of time that working with media can become very complicated and intricate. For that reason you'll probably find it helpful to refer to some of the topics in the documentation, such as [Processing image files](#), [Transcoding](#), and [Using media extensions](#).

Media File Metadata

With a `StorageFolder` in hand for some media library or subset thereof, you can use, as we also saw in Chapter 8, its `getItemsAsync` method to retrieve its contents. You can also use file queries to enumerate those files that match specific criteria. Whatever the case, you end up with a collection of `StorageFile` objects that you can work with however you want.

Now comes the interesting part. As I mentioned in Chapter 8, you can retrieve additional metadata for those files. This has a number of layers that you discover when you start opening some of the secrets doors of the `StorageFile` class, as illustrated in Figure 10-3. The following sections discuss these areas in turn.

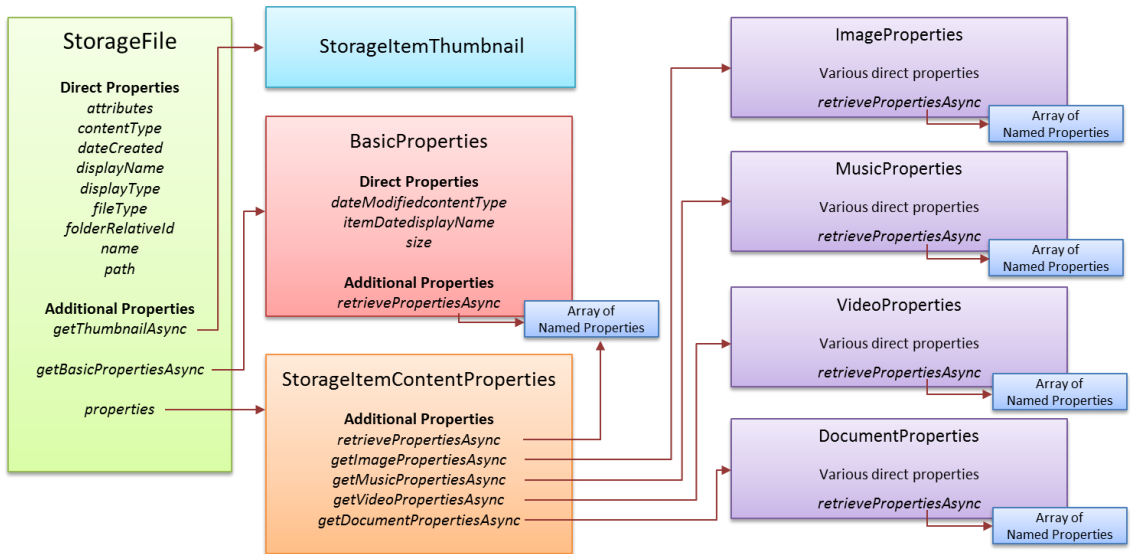


FIGURE 10-3 Relationships between the `StorageFile` object and others obtainable through it.

Thumbnails

First, `StorageFile.getThumbnailAsync` method provides a thumbnail image appropriate for a particular “mode” from the `Windows.Storage.FileProperties.ThumbnailMode` enumeration. Options here are `picturesView`, `videosView`, `musicView`, `documentsView`, `listView`, and `singleItem`. What you get back from this method (through the completed handler when you call `done` on the async promise) is a `StorageItemThumbnail` object that provides thumbnail data as a stream that you can conveniently pass to our old friend `URL.createObjectURL` for display in an `img` element and whatnot.

Examples of this are found throughout the [Retrieve thumbnails for files and folders sample](#). Scenario 1, for instance (`js/scenario1.js`), obtains the thumbnail and displays it in an `img` element:

```
file.getThumbnailAsync(thumbnailMode, requestedSize, thumbnailOptions).done(function (thumbnail) {
    if (thumbnail) {
        outputResult(file, thumbnail, modeNames[modeSelected], requestedSize);
    }
    // ...
});

function outputResult(item, thumbnailImage, thumbnailMode, requestedSize) {
    document.getElementById("picture-thumb-imageHolder").src = URL.createObjectURL(thumbnailImage,
        { oneTimeOnly: true });
    // ...
}
```

Common File Properties

Common file properties—those that exist on all files—are found in a number of different places. Very

common properties are found on the `StorageFile` object directly, like `attributes`, `contentType`, `dateCreated`, `displayName`, `displayType`, `fileType`, `name`, and `path`.

The next group is obtained through `StorageFile.getBasicPropertiesAsync`. This gives you a `Windows.Storage.FileProperties.BasicProperties` object that contains `dateModified`, `itemDate`, and `size` properties. That's a snoozer, you're saying to yourself! Well, this object also has an additional method called `retrievePropertiesAsync` method that gives you an array of name-value pairs for all kinds of other stuff.

The trick to understand here is that you have to give an array of the property names you want to `retrievePropertiesAsync` where each name is a string that comes from a very extensive list of `Windows Properties`, such as `System.FileOwner` and `System.FileAttributes`. An example of this is given in Scenario 5 of the `File Access sample` we saw in Chapter 8:

```
var dateAccessedProperty = "System.DateAccessed";
var fileOwnerProperty    = "System.FileOwner";

SdkSample.sampleFile.getBasicPropertiesAsync().then(function (basicProperties) {
    outputDiv.innerHTML += "Size: " + basicProperties.size + " bytes<br />";
    outputDiv.innerHTML += "Date modified: " + basicProperties.dateModified + "<br />";

    // Get extra properties
    return SdkSample.sampleFile.properties.retrievePropertiesAsync([fileOwnerProperty,
        dateAccessedProperty]);
}).done(function (extraProperties) {
    var propValue = extraProperties[dateAccessedProperty];
    if (propValue !== null) {
        outputDiv.innerHTML += "Date accessed: " + propValue + "<br />";
    }
    propValue = extraProperties[fileOwnerProperty];
    if (propValue !== null) {
        outputDiv.innerHTML += "File owner: " + propValue;
    }
});
```

What's very useful about this is that you can first get to just about any property you want in this way (the list of properties has hundreds of options) and then modify the array and call `BasicProperties.savePropertiesAsync`. Voila! You've just updated those properties on the file. A variation of `savePropertiesAsync` also lets you pass a specific array of name-value pairs if you only want to change specific ones.

The third set of properties is found by going through the secret door of `StorageFile.properties`. This contains a `StorageItemContentProperties` object whose `retrievePropertiesAsync` and `savePropertiesAsync` methods are like those we just saw for `BasicProperties`. What's more interesting is that it also has four other methods—`getDocumentPropertiesAsync`, `getImagePropertiesAsync`, `getMusicPropertiesAsync`, and `getVideoPropertiesAsync`—which are how you get to the really specific stuff for individual file types as we'll see next.

Media-Specific Properties

Alongside the [BasicProperties](#) class in the [Windows.Storage.FileProperties](#) namespace we also find those returned by the [StorageFile.properties.get*PropertiesAsync](#) methods: [DocumentProperties](#), [ImageProperties](#), [MusicProperties](#), and [VideoProperties](#). Though we've had to dig deep to find these, they each contain deeper treasure troves of information—and I do mean deep! The tables below summarize each of these in turn. Note that each object type contains a [retrievePropertiesAsync](#) method, like that of [BasicProperties](#), that lets you request additional properties by name that aren't already included in the main properties object. Refer to the links at the top of the table for the references that identify the most relevant Windows properties.

ImageProperties	from StorageFile.properties.getImagePropertiesAsync	
Additional properties	System.Image , System.Photo , System.Media	
Property	DataType	Applicable Windows Property
title	String	System.Title
dateTaken	Date	System.Photo.DateTaken
latitude	Double (see below)	System.GPS.LatitudeDecimal, or combination of System.GPS.Latitude, System.GPS.LatitudeDenominator, System.GPS.LatitudeNumerator, and System.GPS.LatitudeRef
longitude	Double (see below)	System.GPS.LongitudeDecimal, or combination of System.GPS.Longitude, System.GPS.LongitudeDenominator, System.GPS.LongitudeNumerator, and System.GPS.LongitudeRef
cameraManufacturer	String	System.Photo.CameraManufacturer
cameraModel	String	System.Photo.CameraModel
width	Number in pixels	System.Image.HorizontalSize
height	Number in pixels	System.Image.VerticalSize
orientation	Windows.Storage.FileProperties.PhotoOrientation containing unspecified , normal , flipHorizontal , flipVertical , transpose , transverse , rotate90 , rotate180 , rotate270	System.Photo.Orientation
peopleNames	String vector	System.Photo.PeopleNames
keywords	String vector	System.Keywords
rating	Number (1-99 with 0 meaning "no rating")	System.Rating

VideoProperties	from StorageFile.properties.getVideoPropertiesAsync	
Additional properties	System.Video , System.Media , System.Image , System.Photo	
Property	DataType	Applicable Windows Property
title	String	System.Title
subtitle	String	System.Media.SubTitle
year	Number	System.Media.Year
publisher	String	System.Media.Publisher
rating	Number	System.Rating
width	Number in pixels	System.Video.FrameWidth

height	Number in pixels	System.Video.FrameHeight
orientation	Windows.Storage.FileProperties.VideoOrientation containing normal , rotate90 , rotate180 , rotate270	System.Photo.Orientation
duration	Number (in 100ns units, i.e. 1/10 th milliseconds)	System.Media.Duration
bitrate	Number (in bits/second)	System.Video.TotalBitrate, System.Video.EncodingBitrate
directors	String vector	System.Video.Director
producers	String vector	System.Media.Producer
writers	String vector	System.Media.Writer
keywords	String vector	System.Keywords
latitude	Double (see below)	System.GPS.LatitudeDecimal, or combination of System.GPS.Latitude, System.GPS.LatitudeDenominator, System.GPS.LatitudeNumerator, and System.GPS.LatitudeRef
longitude	Double (see below)	System.GPS.LongitudeDecimal, or combination of System.GPS.Longitude, System.GPS.LongitudeDenominator, System.GPS.LongitudeNumerator, and System.GPS.LongitudeRef

MusicProperties	from StorageFile.properties.getMusicPropertiesAsync	
Additional properties	System.Music , System.Media	
Property	DataType	Applicable Windows Property
title	String	System.Title, System.Music.AlbumTitle
subtitle	String	System.Media.SubTitle
trackNumber	Number	System.Music.TrackNumber
year	Number	System.Media.Year
publisher	String	System.Media.Publisher
artist	String	System.Music.Artist, System.Music.DisplayArtist
albumArtist	String	System.Music.DisplayArtist (read), System.Music.AlbumArtist (write)
genre	String vector	System.Music.Genre
composers	String vector	System.Music.Composer
conductors	String vector	System.Music.Conductor
rating	Number (1-99 with 0 meaning “no rating”)	System.Rating
duration	Number (in 100ns units, i.e. 1/10 th milliseconds)	System.Media.Duration
bitrate	Number (in bits/second)	System.Video.TotalBitrate, System.Video.EncodingBitrate
producers	String vector	System.Media.Producer
writers	String vector	System.Media.Writer

DocumentProperties	from StorageFile.properties.getDocumentPropertiesAsync	
Additional properties	System	
Property	DataType	Applicable Windows Property
title	String	System.Title
Author	String vector	System.Author
keywords	String vector	System.Keywords

Comments	String	System.Comment
----------	--------	----------------

Two notes about all this. First, the string vectors are, as we’ve seen before, instances of [IVector](#) that provide manipulation methods like [append](#), [insertAt](#), [removeAt](#), and so forth. In JavaScript you can access members of the vector like an array with []’s; just remember that the available methods are more specific.

Second, the [latitude](#) and [longitude](#) properties for images and video are double types but contain degrees, minutes, seconds, and a directional reference. The [Simple Imaging sample](#) (in default.js) contains a helper function to extract the components of these values and convert them into a string:

```
"convertLatLongToString": function (latLong, isLatitude) {
    var reference;

    if (isLatitude) {
        reference = (latLong >= 0) ? "N" : "S";
    } else {
        reference = (latLong >= 0) ? "E" : "W";
    }

    latLong = Math.abs(latLong);
    var degrees = Math.floor(latLong);
    var minutes = Math.floor((latLong - degrees) * 60);
    var seconds = ((latLong - degrees - minutes / 60) * 3600).toFixed(2);

    return degrees + "°" + minutes + "'" + seconds + "\"" + reference;
}
```

To summarize, the sign of the value indicates direction. A positive value for latitude means North, negative means South; for longitude positive means East, negative means West. The whole number portion of the value provides the degrees, and the fractional part contains the number of minutes expressed in base 60. Multiplying this value by 60 is the whole minutes, with the remainder then containing the seconds. It’s odd, but that’s the kind of raw data you get from a GPS that geolocation APIs normally convert for you directly.

Media Properties in the Samples

A few of the samples in the Windows SDK show you how to work with some of the properties described in the last section and how to work with those properties more generally. The [Simple Imaging sample](#), in Scenario 1 (js/scenario1.js), provides the most complete demonstration because you can choose an image file and it will load and display various properties, as shown in Figure 10-4 (where I’ve scrolled down to see all the properties). I can verify that the date, camera make/model, and exposure information are all accurate.

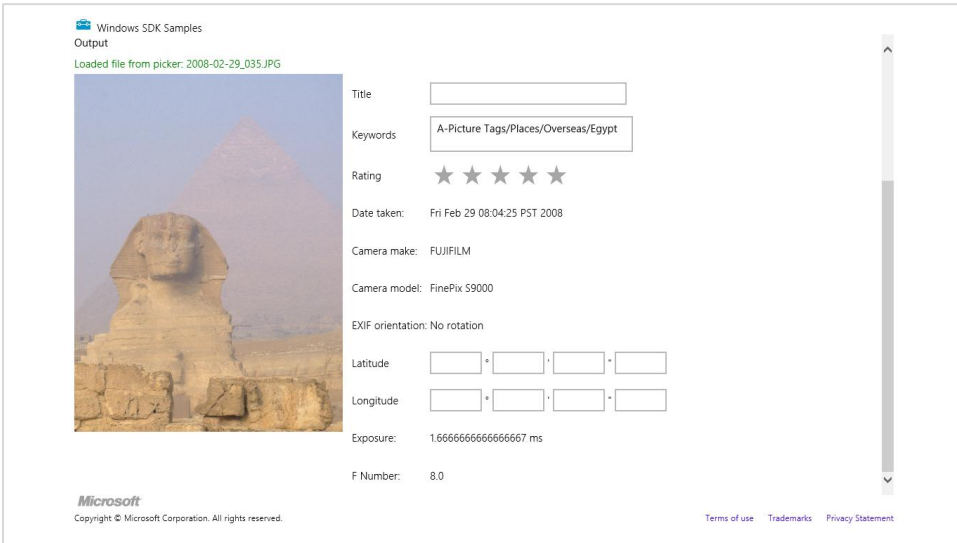


FIGURE 10-4 Image file properties in the Simple Imaging sample.

The `openHandler` method is what retrieves these properties from the file, specifically showing a call to `StorageFile.properties.getImagePropertiesAsync` and the use of `ImageProperties.retrievePropertiesAsync` for a couple of additional properties not already in `ImageProperties`. Then `getImagePropertiesForDisplay` coalesces these into a single object used by the sample's UI. Some lines are omitted from the code shown here:

```
var ImageProperties = {};

function openHandler() {
    // Keep data in-scope across multiple asynchronous methods.
    var file = {};

    Helpers.getFileFromOpenPickerAsync().then(function (_file) {
        file = _file;
        return file.properties.getImagePropertiesAsync();
    }).then(function (imageProps) {
        ImageProperties = imageProps;

        var requests = [
            "System.Photo.ExposureTime",      // In seconds
            "System.Photo.FNumber"             // F-stop values defined by EXIF spec
        ];

        return ImageProperties.retrievePropertiesAsync(requests);
    }).done(function (retrievedProps) {
        // Format the properties into text to display in the UI.
        displayImageUI(file, getImagePropertiesForDisplay(retrievedProps));
    });
}
```



```

function getImagePropertiesForDisplay(retrievedProps) {
    // If the specified property doesn't exist, its value will be null.
    var orientationText = Helpers.getOrientationString(ImageProperties.orientation);

    var exposureText = retrievedProps.lookup("System.Photo.ExposureTime") ?
        retrievedProps.lookup("System.Photo.ExposureTime") * 1000 + " ms" : "";

    var fNumberText = retrievedProps.lookup("System.Photo.FNumber") ?
        retrievedProps.lookup("System.Photo.FNumber").toFixed(1) : "";

    // Omitted: Code to convert ImageProperties.latitude and ImageProperties.longitude to
    // degrees, minutes, seconds, and direction

    return {
        "title": ImageProperties.title,
        "keywords": ImageProperties.keywords, // array of strings
        "rating": ImageProperties.rating, // number
        "dateTaken": ImageProperties.dateTaken,
        "make": ImageProperties.cameraManufacturer,
        "model": ImageProperties.cameraModel,
        "orientation": orientationText,
        // Omitted: lat/long properties
        "exposure": exposureText,
        "fNumber": fNumberText
    };
}

```

Most of the [displayImageUI](#) function to which these properties are passed just copies the data into various controls. It's good to note again, though, that displaying the picture itself is easily accomplished with our good friend, [URL.createObjectURL](#):

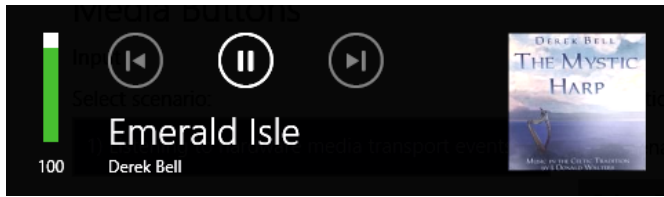
```

function displayImageUI(file, propertyText) {
    id("outputImage").src = window.URL.createObjectURL(file, { oneTimeOnly: true });
}

```

For [MusicProperties](#) a small example can be found in the [Playlist sample](#), as we already saw earlier in "Playlists." You might go back now and look at the code listed in that section, as you should be able to understand what's going on. And while the SDK lacks samples that use [VideoProperties](#) and [DocumentProperties](#), working with these follows the same pattern as shown above for [ImageProperties](#) above, so it should be straightforward to write the necessary code.

Also take a look at the [Configure Keys for Media sample](#), which works with all the different events of the [Windows.Media.MediaControl](#) object and also sets properties like [albumArt](#), [trackName](#), and [artistName](#). The latter properties in particular supply information to the system volume flyout that appears as follows (try changing your system volume to see it, or use one of the hardware media keys on your device, if present):



As for saving properties, the Simple Imaging sample delivers there as well, also in Scenario 1. As the fields shown earlier in Figure 10-4 are editable, the sample provides an Apply button that invokes the `applyHandler` function below to write them back to the file:

```
function applyHandler() {
    ImageProperties.title = id("propertiesTitle").value;

    // Keywords are stored as an array of strings. Split the textarea text by newlines.
    ImageProperties.keywords.clear();
    if (id("propertiesKeywords").value != "") {
        var keywordsArray = id("propertiesKeywords").value.split("\n");

        keywordsArray.forEach(function (keyword) {
            ImageProperties.keywords.append(keyword);
        });
    }

    var properties = new Windows.Foundation.Collections.PropertySet();

    // When writing the rating, use the "System.Rating" property key.
    // ImageProperties.rating does not handle setting the value to 0 (no stars/unrated).
    properties.insert("System.Rating", Helpers.convertStarsToSystemRating(
        id("propertiesRatingControl").winControl.userRating
    ));

    // Code omitted: convert discrete latitude/longitude values from the UI into the
    // appropriate forms needed for the properties, and do some validation; the end result
    // is to store these in the properties list
    properties.insert("System.GPS.LatitudeRef", latitudeRef);
    properties.insert("System.GPS.LongitudeRef", longitudeRef);
    properties.insert("System.GPS.LatitudeNumerator", latNum);
    properties.insert("System.GPS.LongitudeNumerator", longNum);
    properties.insert("System.GPS.LatitudeDenominator", latDen);
    properties.insert("System.GPS.LongitudeDenominator", longDen);

    // Write the properties array to the file
    ImageProperties.savePropertiesAsync(properties).done(function () {
        // ...
    }, function (error) {
        // Some error handling as some properties may not be supported by all image formats.
    });
}
```

A few noteworthy features of this code include the following:

- It separates keywords in the UI control and separately appends each to the keywords

property vector.

- It creates a new collection of properties of type [Windows.Foundation.Collections.PropertySet](#) and uses its insert method to add properties to the list. This property set is what's expected by the [savePropertiesAsync](#) method.
- The [Helpers.convertStartsToSystemRating](#) method (see `default.js`) converts between 1–5 stars, as used in the [WinJS.UI.Rating](#) control, to the *System.Rating* value that uses a 1–99 range. The documentation for [System.Rating](#) specifically indicates this mapping.

In general, all the detailed information you want for any particular Windows property can be found on the reference page for that property. Again start at the [Windows Properties](#) and drill down from there.

Image Manipulation and Encoding

To do something more with an image than just loading and displaying it (where again you can apply various CSS transforms for effect), you need to get to the actual pixels by means of a *decoder*. This already happens under the covers when you assign a URL to an [img.src.](#), but to have direct access to pixels means decoding manually. On the flip side, saving pixels back out to an image file means using an encoder.

WinRT provides APIs for both in the [Windows.Graphics.Imaging](#) namespace, namely in the [BitmapDecoder](#), [BitmapTransform](#), and [BitmapEncoder](#) classes. Loading, manipulating, and saving an image file often involves these three classes in turn, though the [BitmapTransform](#) object is focused on rotation and scaling so you won't use it if you're doing a different manipulation.

One demonstration of this API can be found in Scenario 2 of the [Simple Imaging sample](#). I'll leave it to you to look at the code directly, however, because it gets fairly involved—up to 11 chained promises to save a file! It also does all decoding, manipulation, and encoding within a single function such as [saveHandler](#) (`scenario2.js`). Here's the process it follows:

- Open a file with [StorageFile.openAsync](#), which provides a stream.
- Pass that stream to the static method [BitmapDecoder.createAsync](#) which provides a specific instance of [BitmapDecoder](#) for the stream.
- Pass that decoder to the static method [BitmapEncoder.createForTranscodingAsync](#), which provides a specific [BitmapEncoder](#) instance. This encoder is created with an [InMemoryRandomAccessStream](#).
- Set properties in the encoder's [bitmapTransform](#) property (a [BitmapTransform](#) object) to set the scaling and rotation. This creates the transformed graphic in the in-memory stream.
- Create a property set ([Windows.Graphics.Imaging.BitmapPropertySet](#)) that includes

System.Photo.Orientation and use the encoder's `bitmapProperties.setPropertyAsync` to save it.

- Copy the in-memory stream to the output file stream by using `Windows.Storage.Stream.RandomAccessStream.copyAsync`.

As comprehensive as this scenario is, it's helpful to look at different stages of the process separately, for which purpose we have the ImageManipulation example in this chapter's companion content. This lets you pick and load an image, convert it to grayscale, and save that converted image to a new file. Its output is shown in Figure 10-5. It also gives us an opportunity to see how we can send decoded image data to an HTML `canvas` element and save that canvas's contents to a file.

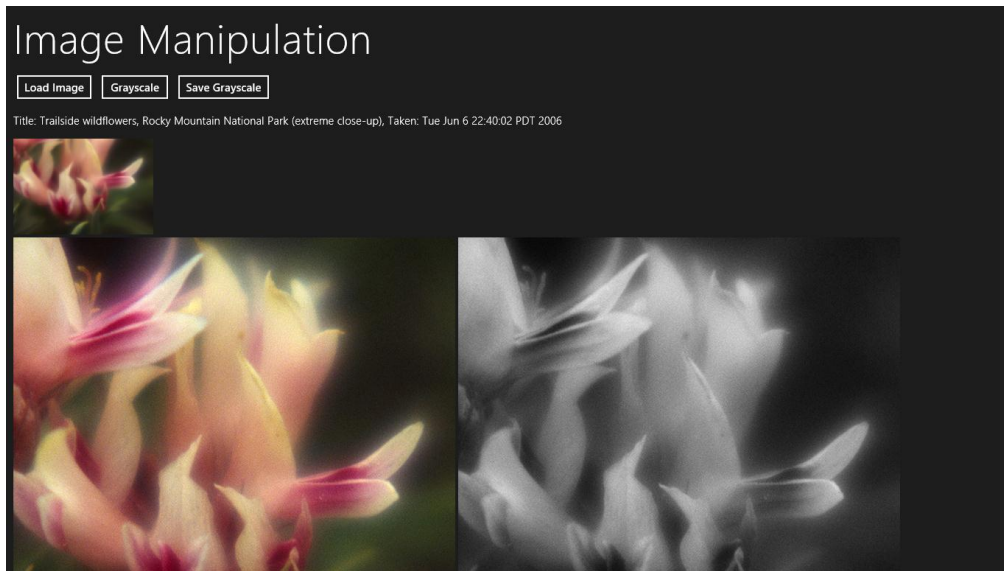


FIGURE 10-5 Output of the ImageManipulation sample in the chapter's companion content.

The handler for the Load Image button (`loadImage` in `default.js`) provides the initial display. It lets you select an image with the file picker, displays the full-size image in an `img` element with `URL.createObjectURL`, calls `StorageFile.properties.getImagePropertiesAsync` to retrieve the `title` and `dateTaken` properties, and uses `StorageFile.getThumbnailAsync` to provide the thumbnail at the top. We've seen all of these APIs in action already.

When we click Grayscale we enter the `setGrayscale` handler where the interesting work happens. We call `StorageFile.openReadAsync` to get a stream, call `BitmapDecoder.createAsync` with that to obtain a decoder, cache some details from the decoder in a local object (`encoding`), and call `BitmapDecoder.getPixelDataAsync` and copy those pixels to a canvas (and only three chained async operations here!):

```
var Imaging = Windows.Graphics.Imaging; //Shortcut
var imageFile;                        //Saved from the file picker
```

```

var decoder; //Saved from BitmapDecoder.createAsync
var encoding = {}; //To cache some details from the decoder

function setGrayscale() {
    //Decode the image file into pixel data for a canvas

    //Get an input stream for the file (StorageFile object saved from opening)
    imageFile.openReadAsync().then(function (stream) {
        //Create a decoder using static createAsync method and the file stream
        return Imaging.BitmapDecoder.createAsync(stream);
    }).then(function (decoderArg) {
        decoder = decoderArg;

        //Configure the decoder if desired. Default is BitmapPixelFormat.rgb8 and
        //BitmapAlphaMode.ignore. You can also use the parameterized version of getPixelDataAsync
        //to control transform, ExifOrientationMode, and ColorManagementMode if needed.

        //Cache these settings for encoding later
        encoding.dpiX = decoder.dpiX;
        encoding.dpiY = decoder.dpiY;
        encoding.pixelFormat = decoder.bitmapPixelFormat;
        encoding.alphaMode = decoder.bitmapAlphaMode;
        encoding.width = decoder.pixelWidth;
        encoding.height = decoder.pixelHeight;

        return decoder.getPixelDataAsync();
    }).done(function (pixelProvider) {
        //detachPixelData gets the actual bits (array can't be returned from an async operation)
        copyGrayscaleToCanvas(pixelProvider.detachPixelData(),
            decoder.pixelWidth, decoder.pixelHeight);
    });
}

```

The decoder's [getPixelDataAsync](#) method comes in two forms. The simple form, shown here, decodes using defaults. The full-control version lets you specify other parameters, as explained in the code comments above. A common use of this is doing a transform using a [Windows.Graphics.Imaging.BitmapTransform](#) object (as mentioned before), which accommodates scaling (with different interpolation modes), rotation (90-degree increments), cropping, and flipping.

Either way, what you get back from the [getPixelDataAsync](#) is not the actual pixel array, because of a limitation in the WinRT language projection mechanism whereby an asynchronous operation cannot return an array. Instead, the operation returns a [PixelDataProvider](#) object whose singular super-exciting synchronous method called [detachPixelData](#) gives you the array you want. (And that method can be called only once and will fail on subsequent calls, hence the “detach” name.) In the end, though, what we have is exactly the data we need to manipulate the pixels and display the result on a canvas, as the [copyGrayscaleToCanvas](#) function demonstrates. You can, of course, replace this kind of function with any other manipulation routine:

```

function copyGrayscaleToCanvas(pixels, width, height) {
    //Set up the canvas context and get its pixel array
    var canvas = document.getElementById("canvas1");
}

```

```

canvas.width = width;
canvas.height = height;
var ctx = canvas.getContext("2d");

//Loop through and copy pixel values into the canvas after converting to grayscale
var imgData = ctx.createImageData(canvas.width, canvas.height);
var colorOffset = { red: 0, green: 1, blue: 2, alpha: 3 };
var r, g, b, gray;

for (var i = 0; i < pixels.length; i += 4) {
    r = pixels[i + colorOffset.red];
    g = pixels[i + colorOffset.green];
    b = pixels[i + colorOffset.blue];

    //Assign each rgb value to brightness for
    gray = Math.floor(.3 * r + .55 * g + .11 * b);

    imgData.data[i + colorOffset.red] = gray;
    imgData.data[i + colorOffset.green] = gray;
    imgData.data[i + colorOffset.blue] = gray;
    imgData.data[i + colorOffset.alpha] = pixels[i + colorOffset.alpha];
}

//Show it on the canvas
ctx.putImageData(imgData, 0, 0);

//Enable save button
document.getElementById("btnSave").disabled = false;
}

```

This is a great place to point out that JavaScript isn't necessarily the best language for working over a pile of pixels like this. (The example program could really use a progress indicator!) Such routines are best implemented as a WinRT component in a language like C++ and made callable by JavaScript. In fact, we'll take the opportunity to do exactly this in Chapter 16, "WinRT Components."

Saving this canvas data to a file then happens in the `saveGrayscale` function, where we use the file picker to get a `StorageFile`, open a stream, acquire the `canvas` pixel data, and hand it off to a `BitmapEncoder`:

```

function saveGrayscale() {
    var picker = new Windows.Storage.Pickers.FileSavePicker();
    picker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.picturesLibrary;
    picker.suggestedFileName = imgData.name + " - grayscale";
    picker.fileTypeChoices.insert("PNG file", [".png"]);

    var imgData, fileStream = null;

    picker.pickSaveFileAsync().then(function (file) {
        if (file) {
            return file.openAsync(Windows.Storage.FileAccessMode.readWrite);
        } else {
            return WinJS.Promise.wrapError("No file selected");
        }
    })
}

```

```

}).then(function (stream) {
    fileStream = stream;
    var canvas = document.getElementById("canvas1");
    var ctx = canvas.getContext("2d");
    imgData = ctx.getImageData(0, 0, canvas.width, canvas.height);

    return Imaging.BitmapEncoder.createAsync(Imaging.BitmapEncoder.pngEncoderId, stream);
}).then(function (encoder) {
    //Set the pixel data--assume "encoding" object has options from elsewhere.
    //Conversion from canvas data to Uint8Array is necessary because the array type
    //from the canvas doesn't match what WinRT needs here.
    encoder.setPixelData(encoding.pixelFormat, encoding.alphaMode,
        encoding.width, encoding.height, encoding.dpiX, encoding.dpiY,
        new Uint8Array(imgData.data));

    //Go do the encoding
    return encoder.flushAsync();
}).done(function () {
    fileStream.close();
}, function () {
    //Empty error handler (do nothing if the user canceled the picker)
});
});
}

```

Note how the `BitmapEncoder` takes a codec identifier in its first parameter. We're using `pngEncoderId`, which is, as you can see, defined as a static property of the `Windows.Graphics.Imaging.BitmapEncoder` class; other values are `bmpEncoderId`, `gifEncoderId`, `jpegEncoderId`, `jpegXREncoderId`, and `tiffEncoderId`. These are the formats supported by the API. You can set additional properties of the `BitmapEncoder` before setting pixel data, such as its `BitmapTransform`, which will then be applied during encoding.

One gotcha to be aware of here is that the pixel array obtained from a `canvas` element (a DOM `CanvasPixelArray`) is not directly compatible with the WinRT byte array required by the encoder. This is the reason for the `new Uint8Array` call down there in the last parameter.

Transcoding and Custom Image Formats

In the previous section we mostly saw the use of a `BitmapEncoder` created with that class's static `createAsync` method to write a new file. That's all well and good, but you might want to know about a few of the encoder's other capabilities.

First is the `BitmapEncoder.createForTranscodingAsync` method that was mentioned briefly in the context of the Simple Imaging sample. This specifically creates a new encoder that is initialized from an existing `BitmapDecoder`. This is primarily used to manipulate some aspects of the source image file while leaving the rest of the data intact. To be more specific, you can first change those aspects that are expressed through the encoder's `setPixelData` method: the pixel format (`rgba8`, `rgba16`, and `bgra8`, see `BitmapPixelFormat`), the alpha mode (premultiplied, straight, or ignore, see `BitmapAlphaMode`), the image dimensions, the image DPI, and, of course, the pixel data itself. Beyond that, you can change other properties through the encoder's `bitmapProperties.setPropertiesAsync`

method. In fact, if all you need to do is change a few properties and you don't want to affect the pixel data, you can use [BitmapEncoder.createForInPlacePropertyEncodingAsync](#) instead (how's that for a method name!). This encoder allows calls to only [bitmapProperties.setPropertiesAsync](#), [bitmapProperties.getPropertiesAsync](#), and [flushAsync](#), and since it can assume that the underlying data in the file will remain unchanged, it executes much faster than its more flexible counterparts and has less memory overhead.

An encoder from [createForTranscodingAsync](#) does *not* accommodate a change of image file format (e.g., JPEG to PNG); for that you need to use [createAsync](#) wherein you can specify the specific kind of encoding. As we've already seen, the first argument to [createAsync](#) is a codec identifier, for which you normally pass one of the static properties on [Windows.Graphics.Imaging.BitmapEncoder](#). What I haven't mentioned yet is that you can also specify custom codecs in this first parameter and that the [createAsync](#) call also supports an optional third argument in which you can provide options for the particular codec in question. However, there are complications and restrictions here.

Let me address options first. You'll find that the present documentation for the [BitmapEncoder](#) codec values (like [pngEncoderId](#)) lacks any details about available options. For that you need to instead refer to the docs for the Windows Imaging Component (WIC), specifically the [Native WIC Codecs](#) that are what the WinRT is surfacing to WinRT apps. If you go into the page for a specific codec, you'll then see a section on "Encoder Options" that tells you what you can use. For example, the [JPEG codec](#) supports properties like [ImageQuality](#) (a value between 0.0 and 1.0), as well as built-in rotations. The [PNG codec](#) supports properties like [FilterOption](#) for various compression optimizations.

To provide these properties, you need to create a new [BitmapPropertySet](#) and insert an entry in that set for each desired options. If, for example, you have a variable named [quality](#) that you want to apply to a JPEG encoding, you'd create the encoder like this:

```
var options = new Windows.Graphics.Imaging.BitmapPropertySet();
options.insert("ImageQuality", quality);
var encoderPromise = Imaging.BitmapEncoder.createAsync(Imaging.BitmapEncoder.jpegEncoderId,
    stream, options);
```

You use the same [BitmapPropertySet](#) for any properties you might pass to an encoder's [bitmapProperties.setPropertiesAsync](#) call. Here's we're just using the same mechanism for encoder options.

As for custom codecs, this simply means that the first argument to [BitmapEncoder.createAsync](#) (as well as [BitmapDecoder.createAsync](#)) is the GUID (a class identifier or CLSID) for that codec, the implementation of which must be provided by a DLL. Details on how to write one of these is provided in [How to Write a WIC-Enabled Codec](#). The catch is that including custom image codecs in your package is not presently supported. If the codec is already on the system (that is, installed via the desktop), it will work. However, the Windows Store policies do not allow WinRT apps that are dependent on other apps, so it's unlikely that you can even ship such an app unless it's preinstalled on some specific OEM device where the DLL is part of the system image. (An app written in C++ can do

more here, but that's well beyond the scope of this book.)

In short, for WinRT apps written in JavaScript and HTML, you're really limited, for all practical purposes, to image formats that are inherently supported in the system.

Do note that these restrictions do *not* exist for custom audio and video codecs. The [Media extensions sample](#) shows how to do this with a custom video codec, as we'll see in the next section.

Manipulating Audio and Video

As with images, if all we want to do is load the contents of a [StorageFile](#) into an audio or video element, we can just pass that [StorageFile](#) to [URL.createObjectUrl](#) and assign the result to a `src` attribute. Similarly, if we want to get at the raw data, we can just use the [StorageFile.openAsync](#) or [openReadAsync](#) methods to obtain a file stream.

To be honest, opening the file is probably the last thing you'd ever want to do in JavaScript with raw audio or video, if even that. While chewing on an image is a marginally acceptable process in the JavaScript environment, churning on audio and especially video is really best done in a highly performant C++ DLL. In fact, many third-party, platform-neutral C/C++ libraries for such manipulations (that you should be able to directly incorporate into such a DLL) are readily available, and in this case you might as well just let the DLL open the file itself!

That said, WinRT *does* provide for transcoding (converting) between different media formats and provides an extensibility model for custom codecs, effects, and scheme handlers. In fact, we've already seen how to apply custom video effects through the [Media extensions sample](#), and the same DLLs can also be used within an encoding process, where all that the JavaScript code really does is glue the right components together (which it's very good at doing). Let's see how this works with transcoding video first and then with custom codecs.

Transcoding

Transcoding both audio and video is accomplished through the [Windows.Media.Transcoding.-MediaTranscoder](#) class, which supports output formats of mp3 and wma for audio, and mp4, wmv, and m4a for video. The transcoding process also allows you to apply effects and to trim start and end times.

Transcoding happens either from one [StorageFile](#) to another or one [RandomAccessStream](#) to another, and in each case happens according to a [MediaEncodingProfile](#). To set up a transcoding operation you call the [MediaTranscoder.prepareFileTranscodeAsync](#) or [prepareStreamTranscodeAsync](#) method, which returns back a [PrepareTranscodeResult](#) object. This represents the operation that's ready to go, but it won't happen until you call that results [transcodeAsync](#) method. In JavaScript, each result is a promise, allowing you to provide completed and progress handlers for a single operation but also allowing you to combine operations with [WinJS.Promise.join](#). That is, this specific control over starting a transcoding operation allows them to be set up and started later, which is useful for batch processing and doing automatic uploads to a service like YouTube while you're sleeping! (And at times like these I've actually pulled ice packs from

my freezer and placed them under my laptop as a poor-man's cooling system....)

The [Transcoding Media sample](#) provides us with a couple of transcoding scenarios. In Scenario 1 (`/js/presets.js`) we can pick a video file, pick a target format, select a transcoding profile, and turn the machine loose to do the job (with progress being reported), as shown in Figure 10-6.

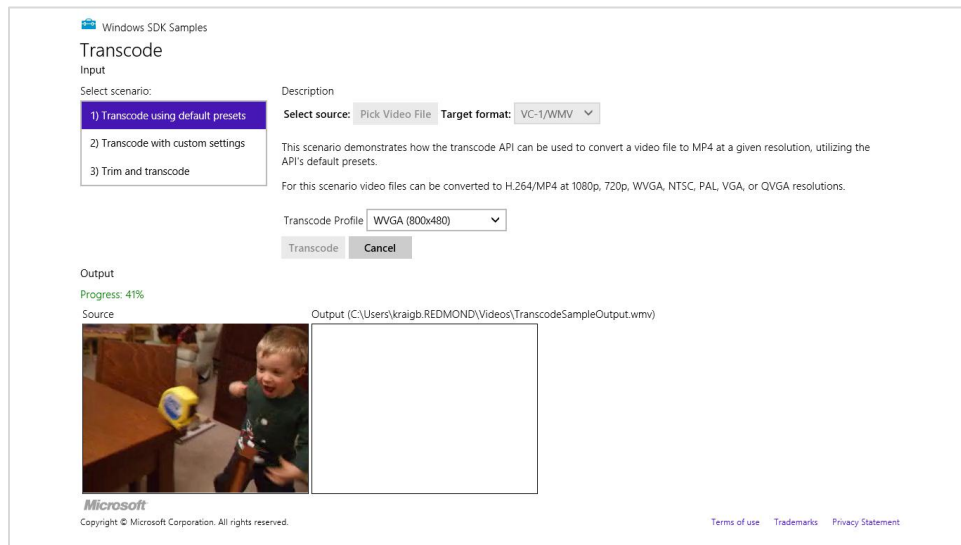


FIGURE 10-6 The Transcoding Media sample cranking away on a video of my then two-year-old son discovering the joys of a tape measure.

The code that's executed when you press the Transcode button is as follows (some bits omitted; this sample happens to use nested promises, which again isn't recommended for proper error handling unless you want, as this code would show, to eat any exceptions that occur prior to the `transcodeAsync` call):

```
function onTranscode() {
    // Create transcode object.
    var transcoder = null;
    transcoder = new Windows.Media.Transcoding.MediaTranscoder();

    // Get transcode profile.
    getPresetProfile(id("profileSelect"));

    // Create output file and transcode.
    var videoLib = Windows.Storage.KnownFolders.videosLibrary;
    var createFileOp = videoLib.createFileAsync(g_outputFileName,
        Windows.Storage.CreationCollisionOption.generateUniqueName);

    createFileOp.done(function (ofile) {
        g_outputFile = ofile;
        g_transcodeOp = null;
        var prepareOp = transcoder.prepareFileTranscodeAsync(g_inputFile, g_outputFile, g_profile);
```

```

prepareOp.done(function (result) {
    if (result.canTranscode) {
        g_transcodeOp = result.transcodeAsync();
        g_transcodeOp.done(transcodeComplete, transcoderErrorHandler, transcodeProgress);
    } else {
        transcodeFailure(result.failureReason);
    }
}); // prepareOp.done
id("cancel").disabled = false;
}); // createFileOp.done
}

```

The `getPresetProfile` method retrieves the appropriate profile object according to the option selected in the app. For the selections shown in Figure 10-6 (WMV and WVGA), we'd use these parts of that function:

```

function getPresetProfile(profileSelect) {
    g_profile = null;
    var mediaProperties = Windows.Media.MediaProperties;
    var videoEncodingProfile;

    switch (profileSelect.selectedIndex) {
        // other cases omitted
        case 2:
            videoEncodingProfile = mediaProperties.VideoEncodingQuality.wvga;
            break;
    }
    if (g_useMp4) {
        g_profile = mediaProperties.MediaEncodingProfile.createMp4(videoEncodingProfile);
    } else {
        g_profile = mediaProperties.MediaEncodingProfile.createWmv(videoEncodingProfile);
    }
}

```

In Scenario 2, the sample always uses the WVGA encoding but allows you to set specific values for the video dimensions, the frame rate, the audio and video bitrates, audio channels, and audio sampling. It applies these settings in `getCustomProfile` (js/custom.js) simply by configuring the profile properties after the profile is created:

```

function getCustomProfile() {
    if (g_useMp4) {
        g_profile = Windows.Media.MediaProperties.MediaEncodingProfile.createMp4(
            Windows.Media.MediaProperties.VideoEncodingQuality.wvga);
    } else {
        g_profile = Windows.Media.MediaProperties.MediaEncodingProfile.createWmv(
            Windows.Media.MediaProperties.VideoEncodingQuality.wvga);
    }

    // Pull configuration values from the UI controls
    g_profile.audio.bitsPerSample = id("AudioBPS").value;
    g_profile.audio.channelCount = id("AudioCC").value;
    g_profile.audio.bitrate = id("AudioBR").value;
}

```

```

g_profile.audio.sampleRate = id("AudioSR").value;
g_profile.video.width = id("VideoW").value;
g_profile.video.height = id("VideoH").value;
g_profile.video.bitrate = id("VideoBR").value;
g_profile.video.frameRate.numerator = id("VideoFR").value;
g_profile.video.frameRate.denominator = 1;
}

```

And to finish off, Scenario 3 is like Scenario 1, but it lets you set start and end times that are then saved in the transcoder's `trimStartTime` and `trimStopTime` properties (see `js/trim.js`):

```

transcoder = new Windows.Media.Transcoding.MediaTranscoder();
transcoder.trimStartTime = g_start;
transcoder.trimStopTime = g_stop;

```

Through not shown in the sample, you can apply effects to a transcoding operation by using the transcoder's `addAudioEffect` and `addVideoEffect` methods.

Custom Decoders/Encoders and Scheme Handlers

Clearly, there are many more audio and video formats in the world than Windows can support in-box, so an extensibility mechanism is provided in WinRT to allow for custom bytestream objects, custom media sources, and custom codecs and effects. It's important to note again that all such extensions are available *only* to the app itself and are not available to other apps on the system. Furthermore, Windows will always prefer in-box components over a custom one, which means don't bother wasting your time creating a new mp3 decoder or such since it will never actually be used!

As suggested earlier with custom image formats, this subject will certainly take you into some pretty vast territory around the entire [Windows Media Foundation \(WMF\) SDK](#). What's in WinRT is really just a wrapper, so knowledge of WMF is essential and not for the faint of heart!

Audio and video extensions are declared in the app manifest where you'll need to edit the XML directly. As seen in the [Media extensions sample](#) for all the DLLs in its overall solution, each declaration looks like this:

```

<Extension Category="windows.activatableClass.inProcessServer">
  <InProcessServer>
    <Path>MPEG1Decoder.dll</Path>
    <ActivatableClass ActivatableClassId="MPEG1Decoder.MPEG1Decoder" ThreadingModel="both" />
  </InProcessServer>
</Extension>

```

The `ActivatableClassId` is how an extension is identified in when calling the WinRT APIs, which is clearly mapped in the manifest to the specific DLL that needs to be loaded.

Depending, then, on the use of the extension, you might need to register them with WinRT through the methods of [Windows.Media.MediaExtensionManager](#): `registerAudio[Decoder | Encoder]`, `registerByteStreamHandler` (media sinks), `registerSchemeHandler` (media sources/file containers), and `registerVideo[Decoder | Encoder]`. In Scenario 1 of the Media extensions sample (LocalDecoder.js), we can see how to set up a custom decoder for video playback:

```

var page = WinJS.UI.Pages.define("/html/LocalDecoder.html", {
    extensions: null,
    MFVideoFormat_MPG1: { value: "{3147504d-0000-0010-8000-00aa00389b71}" },
    NULL_GUID: { value: "{00000000-0000-0000-0000-000000000000}" },

    ready: function (element, options) {
        if (!this.extensions) {
            // Add any initialization code here
            this.extensions = new Windows.Media.MediaExtensionManager();
            // Register custom ByteStreamHandler and custom decoder.
            this.extensions.registerByteStreamHandler("MPEG1Source.MPEG1ByteStreamHandler",
                ".mpg", null);
            this.extensions.registerVideoDecoder("MPEG1Decoder.MPEG1Decoder",
                this.MFVideoFormat_MPG1, this.NULL_GUID);
        }

        // ...
    }
});

```

where the `MPEG1Source.MPEG1ByteStreamHandler` CLSID is implemented in one DLL (see the `MPEG1Source C++` project in the sample's solution) and the `MPEG1Decoder.MPEG1.Decoder` CLSID is implemented in another (the `MPEG1Decoder C++` project).

Scenario 2, for its part, shows the use of a custom scheme handler, where the handler (in the `GeometricSource C++` project) generates video frames on the fly. Fascinating stuff, but again beyond the scope of this book.

Effects, as we've seen, are quite simple to use once you have one implemented: just pass their ID to methods like `msInsertVideoEffect` and `msInsertAudioEffect` on `video` and `audio` elements. Again, you can also apply effects during the transcoding process in the `MediaTranscoder` class's `addAudioEffect` and `addVideoEffect` methods. The same is also true for media capture, as we'll see shortly.

Media Capture

There are times when we can really appreciate the work that people have done to protect individual privacy, such as making sure I know when my computer's camera is being used since I am often using it in the late evening, sitting in bed, or in the early pre-shower mornings when I have, in the words of my father-in-law, "pineapple head."

And there are times when we want to turn on a camera or a microphone and record something: a picture, a video, or audio. Of course, an app cannot know ahead of time what exact camera and microphones might be on a system. A key step in capturing media, then, is determining which device to use—something that the `Windows.Media.Capture` APIs provide for nicely, along with the process of doing the capture itself into a file, a stream, or some other custom "sink" depending on how an app wants to manipulate or process the capture.

Back in Chapter 2, "Quickstart," we learned how to use the camera capture UI functionality in WinRT

to easily capture a photograph in the Here My Am! app. To quickly review, we only needed to declare the webcam capability in the manifest and add a few lines of code:

```
function capturePhoto() {
    var that = this;

    var captureUI = new Windows.Media.Capture.CameraCaptureUI();

    //Indicate that we want to capture a PNG that's no bigger than our target element --
    //the UI will automatically show a crop box of this size
    captureUI.photoSettings.format = Windows.Media.Capture.CameraCaptureUIPhotoFormat.png;
    captureUI.photoSettings.croppedSizeInPixels =
        { width: this.clientWidth, height: this.clientHeight };

    captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
        .done(function (capturedFile) {
            //Be sure to check validity of the item returned; could be null if the user canceled.
            if (capturedFile) {
                lastCapture = capturedFile; //Save for Share
                that.src = URL.createObjectURL(capturedFile, {oneTimeOnly: true});
            }
        }, function (error) {
            console.log("Unable to invoke capture UI.");
        });
}
```

The UI that Windows brings up through this API provides for cropping, retakes, and adjusting camera settings. Another example of taking a photo can also be found in Scenario 1 of the [CameraCaptureUI Sample](#), along with an example of capturing video in Scenario 2. In this case (js/capturevideo.js) we configure the capture UI object for a video format and indicate a video mode in the call to `captureFileAsync`:

```
function captureVideo() {
    var dialog = new Windows.Media.Capture.CameraCaptureUI();
    dialog.videoSettings.format = Windows.Media.Capture.CameraCaptureUIVideoFormat.mp4;

    dialog.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.video).done(function (file) {
        if (file) {
            var videoBlobUrl = URL.createObjectURL(file, {oneTimeOnly: true});
        } else {
            //...
        }
    }, function (err) {
        //...
    });
}
```

It should be noted that the webcam capability in the manifest applies only to the image or video side of camera capture. If you want to capture audio, be sure to also select the Microphone capability on the Capabilities tab of the manifest editor.

If you look in the `Windows.Media.Capture.CameraCaptureUI` object, you'll also see many other

options you can configure. Its [photoSettings](#) property, a [CameraCaptureUIPhotoCaptureSettings](#) object, lets you indicate cropping size and aspect ratio, format, and maximum resolution. Its [videoSettings](#) property, a [CameraCaptureUIVideoCaptureSettings](#) object, lets you set the format, set the maximum duration and resolution, and indicate whether the UI should allow for trimming. All useful stuff! You can find discussions of some of these in the docs on [Capturing or rendering audio, video, and images](#), including coverage of managing calls on a Bluetooth device.

Flexible Capture with the MediaCapture Object

Of course, the default capture UI won't necessarily suffice in every use case. For one, it always sends output to a file, but if you're writing a communications app, for example, you'd rather send captured video to a stream or send it over a network without any files involved at all. You might also want to preview a video before any capture actually happens. Furthermore, you may want to add effects during the capture, apply rotation, and perhaps apply a custom encoding.

All of these capabilities are available through the [Windows.Media.Capture.MediaCapture](#) class:

Properties	Description (classes are in the Windows.Media.Capture namespace unless note)
audioController	An AudioDeviceController that controls volume and provides the ability to manage other arbitrary properties that affect the audio stream.
mediaCaptureSettings	A MediaCaptureSettings that contains device IDs and mode settings, and lets you set the source (audio, videoPreview, photo).
videoController	A VideoDeviceController that controls picture properties (brightness, hue, pan/tilt, zoom, etc.). provides adjustments for backlight and AC power frequency, and provides the ability to manage other arbitrary properties that affect the video stream.
Events	Description
failed	Fired when an error occurs during capture.
recordLimitationExceeded	Fired when the user tried to record video or audio past the allowable duration.
Methods	Description
initializeAsync	Initialize the MediaCapture object (with defaults or with a MediaCaptureInitializationSettings object that contains the same stuff as MediaCaptureSettings).
addEffectAsync	Applies an effect.
clearEffectsAsync	Clears all current effects.
capturePhotoToStorageFileAsync capturePhotoToStreamAsync	Captures an image to a storage file or a random access stream. Both take an instance of ImageEncodingProperties to control format (JPEG or PNG), type, dimensions, and other arbitrary Windows Properties as described earlier in the section "Common File Properties."
getEncoderProperty setEncoderProperty	Manages specific encoder properties.
startRecordToStorageFileAsync startRecordToStreamAsync stopRecordAsync	Starts and stops recording to a storage file or random access stream, a MediaEncodingProfile that determines the audio/video format, along with bitrate, quality, video dimensions, etc.

<code>getRecordRotation</code> <code>setRecordRotation</code>	For videos, these manage a <code>VideoRotation</code> value (90-degree increments) to apply to the recording. These do not affect audio.
<code>startRecordToCustomSinkAsync</code>	Starts recording into a custom sink that's described either by an implementation of <code>Windows.Media.IMediaExtension</code> or by an ID plus a property set of settings.
<code>startPreviewAsync</code> <code>startPreviewToCustomSinkAsync</code> <code>stopPreviewAsync</code> <code>getPreviewRotation</code> <code>setPreviewRotation</code>	Same as recording but works for previews. In this case, if you call <code>URL.createObjectURL</code> and pass the <code>MediaCapture</code> object as the first parameter, the result can be assigned to the <code>src</code> attribute of a <code>video</code> element and the preview shows in that element when you call the <code>video.play</code> method.
<code>getPreviewMirroring</code> <code>setPreviewMirroring</code>	Controls preview mirroring, which means to flip the preview horizontally; this accounts for differences in camera direction which can be in the same direction as the user (rear-mounted camera as on a tablet computer), or the opposite direction (camera mounted on a monitor or built into a laptop display).

For a very simple demonstration of previewing video in a `video` element we can look at the [CameraOptionsUI sample](#) in `js/showoptionsui.js`. When you tap the Start Preview button, it creates an initializes a `MediaCapture` object as follows:

```
function initializeMediaCapture() {
    mediaCaptureMgr = new Windows.Media.Capture.MediaCapture();
    mediaCaptureMgr.initializeAsync().done(initializeComplete, initializeError);
}
```

where the `initializeComplete` handler calls into `startPreview`:

```
function startPreview() {
    document.getElementById("previewTag").src = URL.createObjectURL(mediaCaptureMgr);
    document.getElementById("previewTag").play();
    startPreviewButton.disabled = true;
    showSettingsButton.style.visibility = "visible";
    previewStarted = true;
}
```

The other little bit shown in this sample is invoking the `Windows.Media.Capture.CameraOptionsUI`, which happens when you tap its Show Settings button; see Figure 10-7. This is just a system-provided flyout with options that are relevant to the current media stream being captured:

```
function showSettings() {
    if (mediaCaptureMgr) {
        Windows.Media.Capture.CameraOptionsUI.show(mediaCaptureMgr);
    }
}
```

By the way, if you have trouble running a sample like this in the Visual Studio simulator—specifically, you see exceptions when trying to turn on the camera—try running on the local machine or a remote machine instead.

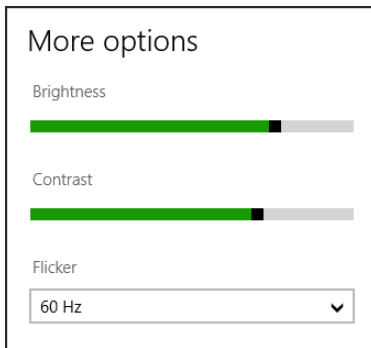


FIGURE 10-7 The Camera Options UI, as shown in the CameraOptionsUI sample (empty bottom is cropped).

More complex scenarios involving the [MediaCapture](#) class (and a few others) can be found now in the [Media capture using capture device sample](#), such as previewing and capturing video, changing properties dynamically (Scenario 1), selecting a specific media device (Scenario 2), and recording just audio (Scenario 3).

Starting with the latter (the simplest), here's the core code to create and initialize the [MediaCapture](#) object for an audio stream (see the [streamingCaptureMode](#) property in the initialization settings) that's directed to a file in the music library via [startRecordToStorageFileAsync](#) (some code omitted for brevity):

```
var mediaCaptureMgr = null;
var captureInitSettings = null;
var encodingProfile = null;
var storageFile = null;

// This is called when the page is loaded
function initCaptureSettings() {
    captureInitSettings = new Windows.Media.Capture.MediaCaptureInitializationSettings();
    captureInitSettings.audioDeviceId = "";
    captureInitSettings.videoDeviceId = "";
    captureInitSettings.streamingCaptureMode = Windows.Media.Capture.StreamingCaptureMode.audio;
}

function startDevice() {
    mediaCaptureMgr = new Windows.Media.Capture.MediaCapture();

    mediaCaptureMgr.initializeAsync(captureInitSettings).done(function (result) {
        // ...
    });
}

function startRecord() {
    // ...
    // Start recording.
    Windows.Storage.KnownFolders.videosLibrary.createFileAsync("cameraCapture.m4a",
        Windows.Storage.CreationCollisionOption.generateUniqueName).done(function (newFile) {
        storageFile = newFile;
    });
}
```

```

        encodingProfile = Windows.Media.MediaProperties.MediaEncodingProfile.createM4a(
            Windows.Media.MediaProperties.AudioEncodingQuality.auto);
        mediaCaptureMgr.startRecordToStorageFileAsync(encodingProfile, storageFile)
        .done(function (result) {
            // ...
        });
    });
}

function stopRecord() {
    mediaCaptureMgr.stopRecordAsync().done(function (result) {
        displayStatus("Record Stopped. File " + storageFile.path + " ");

        // Playback the recorded audio
        var audio = id("capturePlayback" + scenarioId);
        audio.src = URL.createObjectURL(storageFile, { oneTimeOnly: true });
        audio.play();
    });
}

```

Scenario 1 looks very much the same for a video stream as well as photo capture, with results shown in Figure 10-8. Its initialization settings included these properties (see `js/BasicCapture.js` within `initCaptureSettings`):

```

captureInitSettings.photoCaptureSource = Windows.Media.Capture.PhotoCaptureSource.videoPreview;
captureInitSettings.streamingCaptureMode = Windows.Media.Capture.StreamingCaptureMode.audioAndVideo;

```

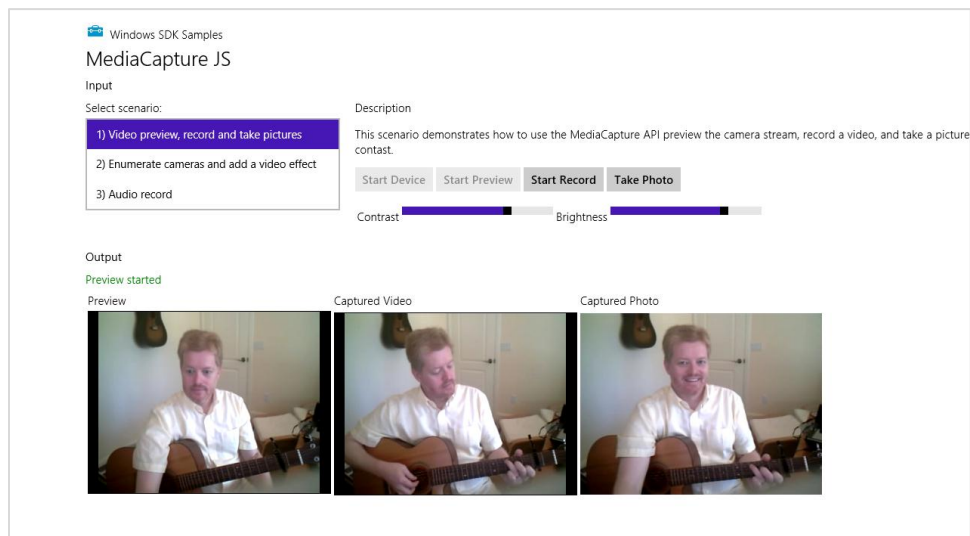


FIGURE 10-8 Previewing and recording video with the default device in the Media capture sample, Scenario 1. (The output is cropped because I needed to run the app using the Local Machine option in Visual Studio, and I didn't think you needed to see a 1920x1200 screenshot with lots of whitespace!).

Notice the Contrast and Brightness controls in Figure 10-8. Changing these will change the preview video, of course, but will also change the recorded video. The sample does this through the

`MediaCapture.videoDeviceController` object's `contrast` and `brightness` properties, showing that these (and any others in the controller) can be adjusted dynamically. Refer to the `getCameraSettings` function in `BasicCapture.js` that basically wires the slider `change` events into a generic anonymous function to update the desired property.

Selecting a Media Capture Device

Scenario 2 now does more or less what Scenario 1 does, but it allows you to select the specific input device. Until now, everything we've done has simply used the default device, but you're not limited to that, of course. You can use the `Windows.Devices.Enumeration` API to retrieve a list of devices within a particular class. In `js/AdvancedCapture.js` we can see how the sample does this for the `videoCapture` class:

```
function enumerateCameras() {
    var cameraSelect = id("cameraSelect");
    deviceList = null;
    deviceList = new Array();
    while (cameraSelect.length > 0) {
        cameraSelect.remove(0);
    }
    //Enumerate webcams and add them to the list
    var deviceInfo = Windows.Devices.Enumeration.DeviceInformation;
    deviceInfo.findAllAsync(Windows.Devices.Enumeration.DeviceClass.videoCapture)
    .done(function (devices) {
        // Add the devices to deviceList
        if (devices.length > 0) {
            for (var i = 0; i < devices.length; i++) {
                deviceList.push(devices[i]);
                cameraSelect.add(new Option(deviceList[i].name), i);
            }
            //Select the first webcam
            cameraSelect.selectedIndex = 0;
            initCaptureSettings();
        } else {
            // disable buttons.
        }
    }, errorHandler);
}
```

The selected device's ID is then copied within `initCaptureSettings` to the `MediaCapture-InitializationSetting.videoDeviceId` property:

```
var selectedIndex = id("cameraSelect").selectedIndex;
captureInitSettings.videoDeviceId = deviceList[selectedIndex].id;
```

By the way, you can retrieve the default device IDs at any time through the methods of the `Windows.Devices.MediaDevice` object and listen to its events for changes in the default devices.

The other bit that Scenario 2 demonstrates is using the `MediaCapture.addEffectAsync` with a

grayscale effect, shown in Figure 10-9, that's implemented in a DLL (the GrayscaleTransform project in the sample's solution). This works exactly as it did with transcoding, and you can refer to the [addRemoveEffect](#) and [addEffectToImageStream](#) functions in `AdvancedCapture.js` for the details. You'll notice there that these functions do a number of checks using the `MediaCaptureSettings.videoDeviceCharacteristic` value to make sure that the effect is added in the right place.

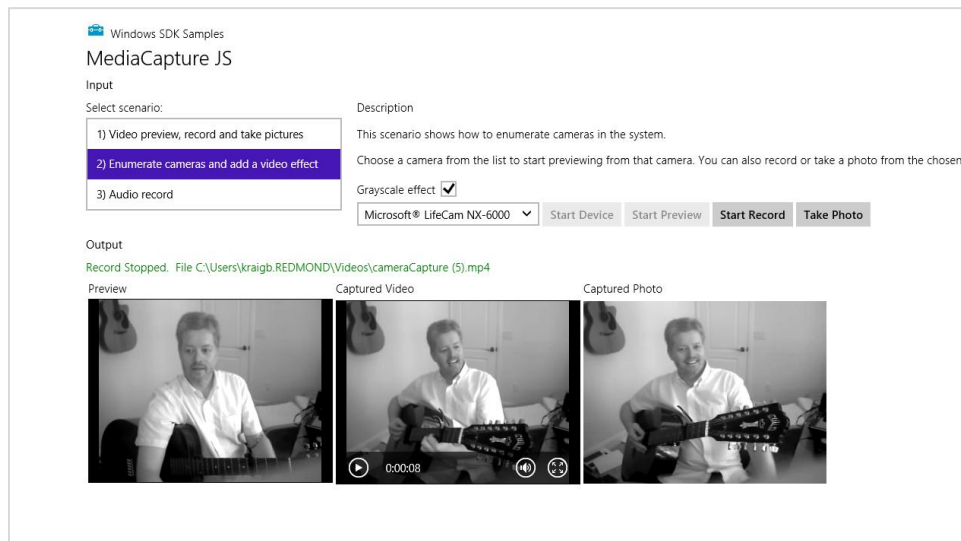


FIGURE 10-9 Scenario 2 of the Media capture sample in which one can select a specific device and apply an effect. (The output here is again cropped from a larger screen shot.) Were you also paying attention enough to notice that I switched guitars?

Streaming Media and PlayTo

To say that streaming media is popular is certainly a gross understatement. As mentioned in this chapter's introduction, Netflix alone consumes for a large percentage of today's Internet bandwidth (including that of my own home). YouTube certainly does its part as well—so your app might as well contribute to the cause!

Streaming media from a server to your app is easily the most common case, and it happens automatically when you set an audio or video `src` attribute to a remote URL. To improve on this, Microsoft also has a [Smooth Streaming SDK for Windows 8 Apps](#) (in beta at the time of writing) that helps you build media apps with a number of rich features including live playback and PlayReady content protection. I won't be covering that SDK in this chapter, but I wanted to make sure you were aware of it.

What we'll focus on here, in the few pages we have left—before my editors at Microsoft Press pull

the plug on this chapter—are considerations for digital rights management (DRM) and streaming not from a network but to a network (for example, audio/video capture in a communications app, as well as streaming media from an app to a PlayTo device).

Streaming from a Server and Digital Rights Management (DRM)

Again, streaming media from a server is what you already do whenever you're using an [audio](#) or [video](#) element with a remote URL. The details just happen for you. Indeed, much of what a great media client app does is talking to web services, retrieving metadata and the catalog, helping the user navigate all of that information, and ultimately getting to a URL that can be dropped in the [src](#) attribute of a [video](#) or [audio](#) element. Then, once the app receives the [canplay](#) event, you can call the element's [play](#) method to get everything going.

Of course, media is often protected with DRM, otherwise the content on paid services wouldn't be generating much income for the owners of those rights! So there needs to be a mechanism to acquire and verify rights somewhere between setting the element's [src](#) and receiving [canplay](#). Fortunately, there's a simple means to do exactly that:

- Before setting the [src](#) attribute, create an instance of [Windows.Media.Protection.MediaProtectionManager](#) and configure its [properties](#).
- Listen to this object's [serviceRequested](#) event, the handler for which performs the appropriate rights checks and sets a completed flag when all is well. (Two other events, just to mention them, are [componentloadfailed](#) and [rebootneeded](#).)
- Assign the protection manager to the audio/video element with the [msSetMediaProtectionManager](#) extension method.
- Set the [src](#) attribute. This will trigger the [serviceRequested](#) event to start the DRM process which will prevent [canplay](#) until DRM checks are completed successfully.
- In the event of an error, the media element's [error](#) event will be fired. The element's [error](#) property will then contain an [msExtendedCode](#) with more details.

You can refer to [How to use pluggable DRM](#) and [How to handle DRM errors](#) for additional details, but here's a minimal and hypothetical example of all this in code:

```
var video1 = document.getElementById("video1");

video1.addEventListener('error', function () {
    var error = video1.error.msExtendedCode;
    //...
}, false);

video1.addEventListener('canplay', function () {
    video1.play();
}, false);

var cpm = new Windows.Media.Protection.MediaProtectionManager();
```

```

cpm.addEventListener('servicerequested', enableContent, false);
video1.msSetContentProtectionManager(cpm);
video1.src = "http://some.content.server.url/protected.wmv";

function enableContent(e) {
    if (typeof (e.request) != 'undefined') {
        var req = e.request;
        var system = req.protectionSystem;
        var type = req.type;

        //Take necessary actions based on the system and type
    }

    if (typeof (e.completion) != 'undefined') {
        //Requested action completed
        var comp = e.completion;
        comp.complete(true);
    }
}

```

How you specifically check for rights, of course, is particular to the service you're drawing from—and not something you'd want to publish in any case!

For a more complete demonstration of handling DRM, check out the [Simple PlayReady sample](#), which will require that you download and install the [Microsoft PlayReady Client SDK](#). PlayReady, if you aren't familiar with it yet, is a license service that Microsoft provides so that you don't have to create one from scratch. The PlayReady client SDK, then, provides additional tools and framework support for apps wanting to implement both online and offline media scenarios, such as progressive download, download to own, rentals, and subscriptions. Plus, with the SDK you don't need to submit your app for DRM Conformance testing. In any case, here's how the Simple PlayReady sample sets up its content protection manager, just to give an idea of how the WinRT APIs are used with specific DRM service identifiers:

```

mediaProtectionManager = new Windows.Media.Protection.MediaProtectionManager();
mediaProtectionManager.properties["Windows.Media.Protection.MediaProtectionSystemId"] =
    '{F4637010-03C3-42CD-B932-B48ADF3A6A54}'

var cpsystems = new Windows.Foundation.Collections.PropertySet();
cpsystems["{F4637010-03C3-42CD-B932-B48ADF3A6A54}"] =
    "Microsoft.Media.PlayReadyClient.PlayReadyWinRTTrustedInput";
mediaProtectionManager.properties["Windows.Media.Protection.MediaProtectionSystemIdMapping"] =
    cpsystems;

```

Streaming from App to Network

The next case to consider is when an app is the source of streaming media rather than the consumer, which means that client apps elsewhere are acting in that capacity. In reality, in this scenario—audio or video communications and conferencing—it's usually the case that the app plays both roles, streaming media to other clients and consuming media from them. This is the case with Windows Live Messenger, Skype, and other such utilities, along with apps like games that include chat capabilities.

Here's how such apps generally work:

- Set up the necessary communication channels over the network, which could be a peer-to-peer system or could involve a central service of some kind.
- Capture audio or video to a stream using the WinRT APIs we've seen (specifically [MediaCapture.startRecordToStreamAsync](#)) or capturing to a custom sink.
- Do any additional processing to the stream data. Note, however, that effects are plugged into the capture mechanism ([MediaCapture.addEffectAsync](#)) rather than something you do in post-processing.
- Encode the stream for transmission however you need.
- Transmit the stream over the network channel.
- Receive transmissions from other connected apps.
- Decode transmitted streams and convert to a blob by using [MSApp.createBlobFromRandomAccessStream](#).
- Use [URL.createObjectURL](#) to hook an [audio](#) or [video](#) element to the stream.

To see such features in action, check out the [Real-time communications sample](#) that implements video chat in Scenario 2 and demonstrates working with different latency modes in Scenario 1. The latter two steps in the list above are also shown in the [PlayToReceiver sample](#) that is set up to receive a media stream from another source.

PlayTo

The final case of streaming is centered on the PlayTo capabilities that were introduced in Windows 7. Simply said, PlayTo is a means through which an app can connect local playback/display for [audio](#), [video](#), and [img](#) elements to a remote device.

The details happen through the [Windows.Media.PlayTo](#) APIs along with the extension methods added to media elements. If, for example, you want to specifically start a process of streaming to a PlayTo device, invoking the selection UI directly, you'd do the following:

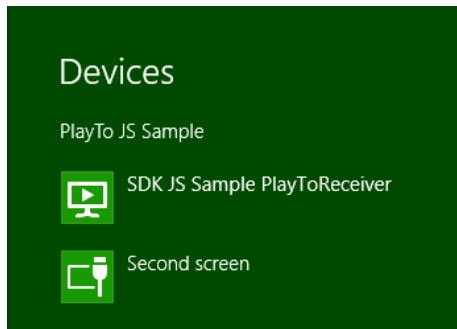
- [Windows.Media.PlayTo.PlayToManager](#):
 - [getForCurrentView](#) returns the object.
 - [showPlayToUI](#) invokes the flyout UI where the user selects a receiver.
 - [sourceRequested](#) event is fired when user selects a receiver.
- In [sourceRequested](#)
 - Get [PlayToSource](#) object from [audio](#), [video](#), or [img](#) element ([msPlayToSource](#) property)

and pass to `e.setSource`.

- Set `PlayToSource.next` property to the `msPlayToSource` of another element for continual playing.
- Pick up `ended` event to stage additional media

Another approach, as demonstrated in the [Media Play To sample](#), is to go ahead and play media locally and then let the user choose a PlayTo device on the fly from the Devices charm. In this case you don't need to do anything because Windows will pick up the current playback element and direct it accordingly. But the app can listen to the `statechanged` event of the element's `msPlayToSource.connection` object (a [PlayToConnection](#)) that will fire when the user selects a PlayTo device and when other changes happen.

Generally speaking, PlayTo is primarily intended for streaming to a media receiver device that's probably connected to a TV or other large screen. This way you can select local content on a Windows 8 device and send it straight to that receiver. But it's also possible to make a software receiver—that is, an app that can receive streamed content from a PlayTo source. The [PlayToReceiver sample](#) does exactly this, and when you run it on another device on your local network, it will show up in the Devices charms as follows:



You can even run the app from your primary machine using the remote debugging tools of Visual Studio, allowing you to step through the code of both source and receiver apps at the same time! (Another option is to run Windows Media Player on one machine and check its Stream > Allow Remote Control of My Player menu option. This should make that machine appear in the PlayTo target list.)

To be a receiver, an app will generally want to declare some additional networking capabilities in the manifest—namely, *Internet (Client & Server)* and *Private Networks (Client & Server)*—otherwise it won't see much action! It then creates an instance of `Windows.Media.PlayTo.PlayToReceiver`, as shown in the PlayTo Receiver sample's `startPlayToReceiver` function (`js/audiovideoptr.js`):

```
function startPlayToReceiver() {
    if (!g_receiver) {
        g_receiver = new Windows.Media.PlayTo.PlayToReceiver();
    }
}
```


Next you'll want to wire up handlers for the element that will play the media stream:

```
var dmrVideo = id("dmrVideo");
dmrVideo.addEventListener("volumechange", g_elementHandler.volumechange, false);
dmrVideo.addEventListener("ratechange", g_elementHandler.ratechange, false);
dmrVideo.addEventListener("loadedmetadata", g_elementHandler.loadedmetadata, false);
dmrVideo.addEventListener("durationchange", g_elementHandler.durationchange, false);
dmrVideo.addEventListener("seeking", g_elementHandler.seeking, false);
dmrVideo.addEventListener("seeked", g_elementHandler.seeked, false);
dmrVideo.addEventListener("playing", g_elementHandler.playing, false);
dmrVideo.addEventListener("pause", g_elementHandler.pause, false);
dmrVideo.addEventListener("ended", g_elementHandler.ended, false);
dmrVideo.addEventListener("error", g_elementHandler.error, false);
```

along with handlers for events that the receiver object will fire:

```
g_receiver.addEventListener("playrequested", g_receiverHandler.playrequested, false);
g_receiver.addEventListener("pauserequested", g_receiverHandler.pauserequested, false);
g_receiver.addEventListener("sourcechangerequested", g_receiverHandler.sourcechangerequested, false);
g_receiver.addEventListener("playbackratechangerequested", g_receiverHandler.playbackratechangerequested, false);
g_receiver.addEventListener("currenttimechangerequested", g_receiverHandler.currenttimechangerequested, false);
g_receiver.addEventListener("mutechangerequested", g_receiverHandler.mutedchangerequested, false);
g_receiver.addEventListener("volumechangerequested", g_receiverHandler.volumechangerequested, false);
g_receiver.addEventListener("timeupdaterequested", g_receiverHandler.timeupdaterequested, false);
g_receiver.addEventListeer("stoprequested", g_receiverHandler.stoprequested, false);
g_receiver.supportsVideo = true;
g_receiver.supportsAudio = true;
g_receiver.supportsImage = false;
g_receiver.friendlyName = 'SDK JS Sample PlayToReceiver';
```

The last line above, as you can tell from the earlier image, is the string that will show in the Devices charm for this receiver once it's made available on the network. This is done by calling `startAsync`:

```
// Advertise the receiver on the local network and start receiving commands
g_receiver.startAsync().then(function () {
    g_receiverStarted = true;

    // Prevent the screen from locking
    if (!g_displayRequest) {
        g_displayRequest = new Windows.System.Display.DisplayRequest();
    }
    g_displayRequest.requestActive();
});
```

Of all the receiver object's events, the critical one is `sourcechangerequested` where `eventArgs.stream` contains the media we want to play in whatever element we choose. This is easily accomplished by creating a blob from the stream and then a URL from the blob that we can assign to an element's `src` attribute:

```
sourcechangerequested: function (eventIn) {
    if (!eventIn.stream) {
        id("dmrVideo").src = "";
    }
}
```

```

    } else {
        var blob = MSApp.createBlobFromRandomAccessStream(eventIn.stream.contentType,
            eventIn.stream);
        id("dmrVideo").src = URL.createObjectURL(blob, {oneTimeOnly: true});
    }
}

```

All the other events, as you can imagine, are primarily for wiring together the source's media controls to the receiver such that pressing a pause button, switching tracks, or acting on the media in some other way at the source will be reflected in the receiver. There may be a lot of events, but handling them is quite simple as you can see in the sample.

What We Have Learned

- Creating media elements can be done in markup or code by using the standard `img`, `svg`, `canvas`, `audio`, and `video` elements.
- The three graphics elements—`img`, `svg`, and `canvas`—can all produce essentially the same output, only with different characteristics as to how they are generated and how they scale. All of them can be styled with CSS, however.
- The `Windows.System.Display.DisplayRequest` object allows for disabling screen savers and the lock screen during video playback (or any other appropriate scenario).
- Both the audio and video elements provide a number of extension APIs (properties, methods, and events) for working with various platform-specific capabilities in Windows 8, such as horizontal mirroring, zooming, playback optimization, 3D video, low-latency rendering, PlayTo, playback management of different audio types or categories, effects (generally provided as DLLs in the app package), and digital rights management.
- Background audio is supported for several categories given the necessary declarations in the manifest and handlers for media control events (so the audio can be appropriately paused and played).
- Through the WinRT APIs, apps can manage very rich metadata and properties for media files, including thumbnails, album art, and properties specific to the media type, including access to a very extensive list of [Windows Properties](#).
- The WinRT APIs provide for decoding and encoding of media files and streams, through which the media can be converted or properties changed. This includes support for custom codecs.
- WinRT provides a rich API for media capture (photo, video, and audio), including a built-in capture UI, along with the ability to provide your own and yet still easily enumerate and access available devices.

- Streaming media is supported from a server (with and without digital rights management, including PlayReady), between apps (inbound and outbound), and from apps to PlayTo devices. An app can also be configured as a PlayTo receiver.

Chapter 11

Purposeful Animations

In the early 1990s, the wonderful world of multimedia first became prevalent on Windows PCs. Before that time it was difficult for such machines to play audio and video, access compact discs (remember those?), and otherwise provide the rich experience we take for granted today. The multimedia experience was so new and exciting, and many people jumped in wholeheartedly, including the group of support engineers at Microsoft who were specializing in this area. Though my team (specializing in UI) sat more than 100 feet away from their area, we could clearly hear—for most of the day!—the various chirps and bleeps emitting from their speakers, against the background of a soft Amazon basin rainfall.

At that time too, many consumers of Windows were having fun attaching all kinds of crazy sounds to every mouse click, window transition, email arrival, and every other system event they could think of. Yet after a month or two of this sensual overload—not unlike being at a busy carnival—most people started to remove quite a few of those sounds, if not disable them altogether. I, for one, eventually turned off all my sounds. Simply said, I got tired of the extra noise.

Along these same lines, you may remember that when DVDs first appeared in their full glory, just about every title had fancy menus with clever transitions. No more: most consumers, I think, got tired of waiting for all this to happen and just want to get on with the business of watching the movie as quickly as possible.

Today we're reliving this same experience with fluid animations. Now that most systems have highly responsive touch screens and GPUs capable of providing very smooth graphical transitions, it's tempting to use animations superfluously. However, unless the animations actually add meaning and function to an app, consumers will likely tire of them like they did with DVD menus, especially if they end up interfering with a workflow by making one constantly wait for the animations to finish! I'll bet that every reader of this book has, at least once, repeatedly hit the Menu button on a DVD remote to no avail....

This is why WinRT app design speaks of *purposeful* animations: if there's no real purpose behind an animation, you should ask, "Why am I wanting to use this?" Take a moment, in fact, to use Windows 8 and some of the built-in apps to explore how animations are both used and *not* used. Notice how many animations are specifically to track or otherwise give immediate feedback for touch interactions, which purposefully help users know that their input is being registered by the system. Other animations, such as when items are added or removed from a list, are intended to draw attention to the change, soften its visual impact, and give it a sense of fluidity. In other cases, you may find apps that perhaps overuse animations, simply using animations because they're available or trying too hard to emulate physical motion where it's simply not necessary. In this way, excessive animations constitute a kind of "chrome" with the same effect as other chrome: distracting the user from the content they

really care about. (If you feel tempted to add little effects that are like this, consider at least providing a setting to turn them off.)

Let me put it another way. When thinking about animations, ask yourself, “What do they communicate?” Animations are a form of communication, a kind of visual language. I would even venture to say (as I am venturing now) that animations really only say one or a combination of three things:

- “Thanks, I heard you,” as when something on the screen moves naturally in response to a user gesture. Without this communication, the user might think that their gesture didn’t register and will almost certainly poke at the app again.
- “Hello” and “Goodbye,” as when items appear or disappear from view, or transition one to another. Without this communication, changes that happen to on-screen elements can be as jarring as Bilbo Baggins slipping on the Ring of Power and instantly vanishing. This is not to say that most consumers are incredulous hobbits, of course, but you get the idea.
- “Hey, look at me!” as when something moves to only gain attention or look cute.

If I were to assign percentages to these categories to represent how often they would or should be used, I’d probably put them at 80%, 15%, and 5%, respectively (although some animations will serve multiple purposes). Put another way, the first bit of communication is really about listening and responding, which is what an app should be doing most of the time. The second bit is about courtesy, which is another good quality to express within reason—courtesy can, like handshakes, hugs, bows, and salutes, be overused to the point of annoyance. The third bit, for its part, can be helpful when there’s a real and sincere reason to raise your hand or offer a friendly wave, but otherwise can easily become just another means of showing off.

There’s another good reason to be judicious about the use of animations and really make them count: power consumption. No matter how it’s accomplished, via GPU or CPU, animation is an expensive process. Every watt of juice in a consumer’s batteries should be directed toward fulfilling their goals with their device rather than scattered to the wind. Again, this is why this chapter is called “*Purposeful Animations*” and not just “*Animations*”!

In any case, you and your designers are the ultimate arbiters of how and when you’ll use animations. In this uncommonly short chapter, then, we’ll first look at what’s provided for you in the WinJS Animations Library, a collection of animations built on CSS that already embody the Windows 8 look and feel for many common operations. After that we’ll review the underlying CSS capabilities that you can, of course, use directly. In fact, aside from games and other apps whose primary content consists of animated objects, you can probably use CSS for most other animation needs. This is a good idea because the CSS engine is very much optimized to take advantage of hardware acceleration, something that isn’t true when doing frame-by-frame animations in JavaScript yourself. Nevertheless, we’ll end this chapter on that latter subject, as there are some tips and tricks for doing it well within WinRT apps.

Systemwide Enabling and Disabling of Animations

Before we go any further, there's a setting buried deep inside the desktop Control Panel's Ease of Access Center that you need to know about because it affects how the WinJS Animations Library behaves and should affect whether you do animations of your own. From the desktop, invoke the Charms and select Control Panel. Then navigate to Ease of Access > Ease of Access Center > Make the computer easier to see. Scroll down close to the bottom and you'll see the item "Turn off all unnecessary animations (when possible)," as shown in Figure 11-1.

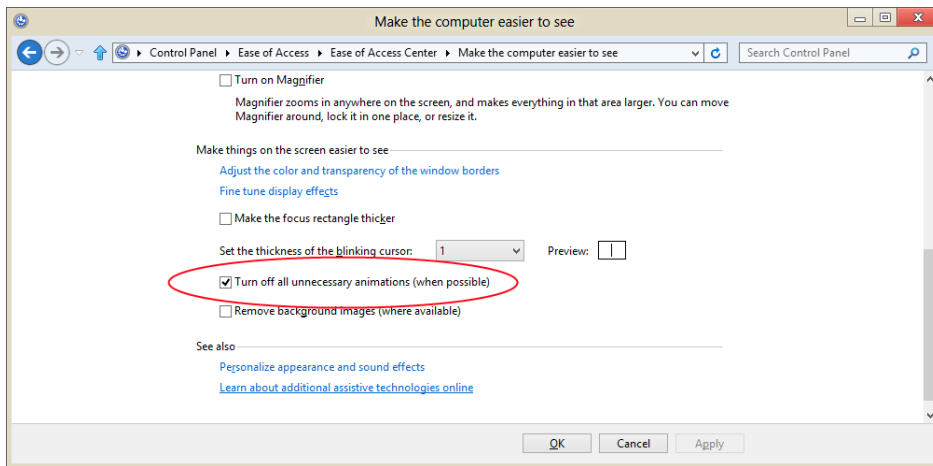


FIGURE 11-1 A very important setting for animation in the desktop control panel.

The idea behind this check box is that for some users, animations are a real distraction that can make the entire machine more difficult to use. For medical reasons too, some users might elect to minimize on-screen movement just to keep the whole experience more calm. So when this option is checked, the WinJS animations don't actually do anything, and it's recommended that apps also disable many if not all of their own custom animations as well.

The Control Panel setting can be obtained through the [Windows.UI.ViewManagement.UISettings](#) class in its `animationsEnabled` property:

```
var settings = new Windows.UI.ViewManagement.UISettings();  
var enabled = settings.animationsEnabled;
```

You can also just call the `WinJS.UI.isAnimationEnabled` method that will return `true` or `false` depending on this property. WinJS obviously uses this internally to manage its own animation behavior.

WinJS also adds an enablement count that you can use to temporarily enable or disable animations in conjunction with the `animationsEnabled` value. You change this count by calling `WinJS.UI.enableAnimations` and `WinJS.UI.disableAnimations`, the effects of which are cumulative,

and the `animationsEnabled` property counts as 0 if the Control Panel option is checked and 1 if it's unchecked.

Thus, if the system setting is checked and you call `WinJS.UI.enableAnimations`, the WinJS animations will execute. If the system setting is not checked and you call `WinJS.UI.disableAnimations`, those animations will not execute.

When implementing your own animations either with CSS or with mechanisms like `setInterval` or `requestAnimationFrame`, it's a good idea to be sensitive to the `animationsEnabled` setting where appropriate. I add this condition because if an animation is essential to the actual content of an app, like a game, then it's not appropriate to apply this setting. The same goes for animating something like a clock within a clock app. It's really about animations that add a fast-and-fluid effect to the content, but it can be turned off without ill effect.

The WinJS Animations Library

When considering animations for your app, the first place you should turn is the Animations Library in WinJS, found in the `WinJS.UI.Animation` namespace. Each animation is basically a function within this namespace that you call when you want a certain kind of animation or transition to happen. The benefit of using these is that they directly embody the Windows 8 look and feel and, in fact, are what WinJS itself uses to animate its own controls, flyouts, and so forth to match the user interface design guidelines. What's more, because they are built with CSS transitions and animations, they aren't dependent on WinRT and are fully functional within web context pages that have loaded WinJS (but they do pay attention to whether animations are enabled as described in the previous section).

All of the animations, as listed in the table below, have guidance as to when and how they should be applied. These are really design questions more than implementation questions. Let me emphasize this: animations should be part of an app's design, not just an implementation detail, because they are very closely related to the overall user experience of an app. I say this because oftentimes app designs are represented with static wireframes or mockups, which don't indicate dynamic elements like animations and transitions. By being aware of what's in the animations library, designers can more readily see where animations are appropriately applied and include them in even the earliest stages of an app design. This is important because animations are best implemented as part of the app's layout, because they are often tightly coupled with layout.

You can find full guidance in the [Animating Your UI](#) and [Animating UI Surfaces](#) topics in the documentation, which will also contain specific guidelines for the individual animations below. I will only summarize here.

Key Point Built-in controls and other UI elements like those we've worked with in previous chapters already make use of the appropriate animations. For example, you don't need to animate a button tap in the `button` element nor animate the appearance or disappearance of controls like `WinJS.UI.Appbar`. You'll primarily use them when implementing UI directly with HTML layout or

when building custom controls.

Animation Name	WinJS.UI.Animation methods	Description and Usage
Page Transition	enterPage , exitPage	Animates a whole page into or out of view, such as when bringing in the first page of an app after the splash screen or when switching between app pages. Avoid using enterPage when content is already on screen—that is, use it only when changing the entirety of the content.
Content Transition	enterContent , exitContent	Animates one or more elements into or out of view, specifically used for content that wasn't ready when a page was loaded or when a section of a page is changing within a container. If other content needs to move in response to the container change, such as if it is resizing, you can move those other elements by using expand/collapse or reposition animations.
Fade In/Out	fadeIn , fadeOut	Used to show or hide transient UI or controls, as is done with scrollbars or when a placeholder is replaced with a loaded item. These are also good default animations for situations where other specific animations don't apply.
Crossfade	crossFade	Softens the transition between different states of an item. This is also used in refresh scenarios, such as when a news app updates all of its content at once.
Pointer Up/Down	pointerUp , pointerDown	Provides immediate feedback for a successful tap or click on an item or "tile." Note that built-in controls like the button and ListView already incorporate these animations.
Expand/Collapse	createExpandAnimation , createCollapseAnimation	Adds or removes extra space within content, such as making room for error messages or hiding an option that isn't needed.
Reposition	createRepositionAnimation	Used when moving an element to a new position.
Show/Hide Popup	showPopup , hidePopup	Used to show and hide popup UI like menus, flyouts, tool tips, and other contextual UI that appears above an app canvas (dialogs, however, use Fade In). Avoid using for elements that are part of that canvas directly—use Content Transition and Fade In/Out animations instead. You also don't need to use these directly when using built-in controls, as those controls already apply the animations.
Show/Hide Edge UI Show/Hide Panel UI	showEdgeUI , hideEdgeUI , showPanel , hidePanel	Used to show and hide edge-oriented UI like app bars and the soft keyboard. The Edge UI animations are for elements that only move a short distance onto the screen; the panel animations are for those that move longer distances. These should not be used for UI that's not moving from or toward an edge; use the Reposition animation

		instead. Crossfade is also typically applied after showing and simultaneous with hiding. The built-in edge controls like the app bar and settings pane already apply these animations.
Peek (for tiles)	<code>createPeekAnimation</code>	Animates a tile update when alternating between showing an image and text when the tile isn't tall enough to see both. Can also be used to cycle through images on a tile. This is the animation used for live tiles on the Windows Start screen.
Badge Update	<code>updateBadge</code>	Used to update the number on a badge.
Swipe Hint	<code>swipeReveal</code>	Used in response to a tap-and-hold event to indicate that an item can be selected with a swipe.
Swipe Select/Deselect	<code>swipeSelect</code> , <code>swipeDeselect</code>	Animates an item when swiped to select or deselect it.
Add/Delete from List	<code>createAddToListAnimation</code> , <code>createDeleteFromListAnimation</code>	Animates the insertion or deletion of items from a list, as used by the <code>ListView</code> control. The add animation repositions existing items to make space for the new and then brings in those new items; the delete animation pulls items out and repositions those that remain. Avoid using these to display or remove a container or to add or remove the entire contents of the collection; use Content Transitions instead.
Add/Delete from Search List	<code>createAddToSearchListAnimation</code> , <code>createDeleteFromSearchListAnimation</code>	These animations are similar to those for adding and removing from a list, but they are designed for much more rapid changes as happens when populating a list of search results. Simply said, they have shorter durations.
Start/End Drag-Drop	<code>dragSourceStart</code> , <code>dragSourceEnd</code> , <code>dragBetweenEnter</code> , <code>dragBetweenLeave</code>	Provides visual feedback during drag-and-drop operations. The start and end animations are for the item being moved and should always be used together; the enter and leave animations are for rearranging the area around a potential drop point, which helps to show how the content will appear if the drop happens. For this purpose you'll need to define the size of potential target areas (rectangles) so that you can track pointer movement in and out of those areas.

If you want to see what these animations are actually doing, you can find all of that in the WinJS source code's `ui.js` file. Just search for the method, and you'll see how they're set up. The Crossfade animation, for example, animates the incoming element's `opacity` property from 0 to 1 over 167ms with a linear timing function, while animating the outgoing element's `opacity` from 1 to 0 in the same way. The Pointer Down animation changes the element's `scale` from 100% to 97.5% over 167ms according to a cubic-bezier curve, while Pointer Up does the opposite.

To be honest, however, as interesting as such details might be, they are always subject to change. Plus, if you really need to obtain such details programmatically, use the APIs in the

[Windows.UI.Core.AnimationMetrics](#) namespace rather than looking at the WinJS sources. And in the end, what's important is that you choose animations not for their visual effects but for their semantic meaning, using the right animations at the right times in the right places. So let's see how we do that.

Tip #1 All of the WinJS animations are implemented using the [WinJS.UI.executeAnimation](#) and [WinJS.UI.executeTransition](#) functions, which you can use for custom animations as well.

Tip #2 While an animation is running always avoid changing an element's contents and its CSS properties that affect the same properties. The results are unpredictable and unreliable and can cause performance problems.

Animations in Action

To see all of the WinJS animations in action, run the [HTML Animation Library sample](#). There are many different animations to illustrate, and this sample most certainly earns the award for the largest number of scenarios: twenty-two! In fact, the first thing you should do is go to Scenario 22 and see whether animations are enabled, as that will most certainly affect your experience with the rest of the same. The output of that scenario will show you whether the [UISettings.animationsEnabled](#) flag is set and allow you to increment or decrement the WinJS enablement count. So go check that now, because if you're like me, you might have turned off system animations a long time ago for a snappier desktop experience. (For example, I dislike waiting for the task bar to animate up and down!) I didn't realize at first that it affected WinJS in this way!

Clearly, with 22 scenarios in the sample I won't be showing code for all of them here; indeed, doing so isn't necessary because many operate in the same way. The only real distinction is between those whose methods start with [create](#) and those that don't, as we'll see in a bit.

All the animation methods return a promise that you can use to take additional action when the animation is complete (at which point your handlers in [then](#) or [done](#) will be called). If you already know something about CSS transitions and animations, you'll rightly guess that these promises encapsulate events like [transitionend](#) and [animationend](#), so you won't need to listen for those events directly if you want to chain or synchronize animations. For chaining, you can just chain the promises; for synchronization, you can obtain the promises for multiple animations and wait for their completion using methods like [WinJS.Promise.join](#) or [WinJS.Promise.any](#).

Animation promises also support the [cancel](#) method, which removes the animation from the element. This immediately sets the affected property values to their final states, causing an immediate visual jump to that end state. And whether you cancel an animation or it ends on its own, the promise is considered to have completed successfully; canceling an animation, in other words, will call the promise's completed handler and not its error handler.

Do be aware that because all of the WinJS animations are implemented with CSS, they won't actually start until you return from whatever function you call the methods and give control back to the UI thread. This means that you can set up multiple animations knowing that they'll more or less

start together once you return from the function. So even though the animation methods return promises, they are not like other asynchronous operations in WinRT that start running on another thread altogether.

Anyway, let's look at some code! In the simplest case, all you need to do is call one of the animation methods and the animation will execute when you return. Scenario 6 of the sample, for instance, just adds these handlers to the `MSPointerDown` and `MSPointerUp` events of three different elements (`js/pointerFeedback.js`):

```
function onPointerDown(evt) {
    WinJS.UI.Animation.pointerDown(evt.srcElement);
}

function onPointerUp(evt) {
    WinJS.UI.Animation.pointerUp(evt.srcElement);
}
```

We typically don't need to do anything when the animations complete, so there's no need for us to call `done` or provide a completed function. Truly, using many of these animations is just this simple. Let's look at some examples.

The `crossFade` animation, for its part (Scenario 10), takes two elements: the incoming element and the outgoing element (all of which must be visible and part of the DOM throughout the animation, mind you!). Calling it then looks like this (`js/crossfade.js`):

```
WinJS.UI.Animation.crossFade(incoming, outgoing);
```

Yet this isn't the whole story. A common feature among the animations is that you can provide an *array* of elements on which to execute the same animation or, in the case of `crossFade`, two arrays of elements. While this isn't useful for animations like `pointerDown` and `pointerUp` (each pointer event should be handled independently), it's certainly handy for most others.

Consider the `enterPage` animation. In its singular form it accepts an element to animate and an optional initial offset where the element begins relative to its final position. (Generally speaking, you should omit this offset if you don't need it, because it will give result in better performance—the sample passes `null` here, which I've omitted in the code below.) `enterPage` can also accept a collection of elements, such as the result of a `querySelectorAll`. So we see how Scenario 1 (`html/enterPage.html` and `js/enterPage.js`) provides a choice of how many elements are animated separately:

```
switch (pageSections) {
    case "1":
        // Animate the whole page together
        enterPage = WinJS.UI.Animation.enterPage(rootGrid);
        break;
    case "2":
        // Stagger the header and body
        enterPage = WinJS.UI.Animation.enterPage([[header, featureLabel], [contentHost, footer]]);
        break;
}
```

```

case "3":
  // Stagger the header, input, and output areas
  enterPage = WinJS.UI.Animation.enterPage([[header, featureLabel], [inputLabel, input],
    [outputLabel, output, footer]]);
  break;
}

```

When the element argument is an array, the offset argument, if provided, can be either a single offset (that is applied to all elements) or an array (to indicate offsets for each element individually). Each offset is an object with the properties that define the offset. See `js/dragBetween.js` for scenario 13 where this is used with the `dragBetweenEnter` animation:

```

WinJS.UI.Animation.dragBetweenEnter([box1, box2],
  [{ top: "-40px", left: "0px" }, { top: "40px", left: "0px" }]);

```

Here's a modification showing a single offset that's applied to both elements:

```

WinJS.UI.Animation.dragBetweenEnter([box1, box2], { top: "0px", left: "40px" });

```

Scenario 4 (`js/transitioncontent.js`) shows how you can chain a couple of promises together to transition between two different blocks of content:⁵⁵

```

WinJS.UI.Animation.exitContent(outgoing, null).done( function () {
  outgoing.style.display = "none";
  incoming.style.display = "block";
  return WinJS.UI.Animation.enterContent(incoming, null);
});

```

Things get a little more interesting when we look at the `create*` animation methods, together referred to as the *layout animations*, which are for adding and removing items from lists, expanding and collapsing content, and so forth. Each of these has a three-step process where you create the animation, manipulate the DOM, and execute the animation, as shown in Scenario 7 (`js/addAndDeleteFromList.js`):

```

// Create addToList animation.
var addToList = WinJS.UI.Animation.createAddToListAnimation(newItem, affectedItems);

// Insert new item into DOM tree.
// This causes the affected items to change position.
list.insertBefore(newItem, list.firstChild);

// Execute the animation.
addToList.execute();

```

The reason for the three-step process is that in order to carry out the animation on newly added items, or items that are being removed, they all need to be in the DOM when the animation executes. The process here lets you create the animation with the initial state of everything, manipulate the DOM

⁵⁵ Note that the actual sample passes the value `output` as the first parameter to `exitContent` and `enterContent`; the code should appear as shown here, with the `outgoing` value being passed to `exitContent` and `incoming` passed to `enterContent`.

(or just set styles and so forth) to create the ending state, and then execute the animation to “let ‘er rip.” You can then use the done method on the promise returned from `execute` to clean do your final cleanup. Scenario 5 (`js/expandAndCollapse.js`) makes this point clear:

```
// Create collapse animation.
var collapseAnimation = WinJS.UI.Animation.createCollapseAnimation(element, affected);

// Remove collapsing item from document flow so that affected items reflow to their new position.
// Do not remove collapsing item from DOM or display at this point, otherwise the animation on the
// collapsing item will not display
element.style.position = "absolute";
element.style.opacity = "0";

// Execute collapse animation.
collapseAnimation.execute().done(
    // After animation is complete (or on error), remove from display.
    function () { element.style.display = "none"; },
    function () { element.style.display = "none"; }
);
```

As a final example—because I know you’re smart enough to look at most of the other cases on your own—Scenario 21 (`js/customAnimation.js`) shows how to use the `WinJS.UI.executeAnimation` and `WinJS.UI.executeTransition` methods.

```
function runCustomShowAnimation() {
    var showAnimation = WinJS.UI.executeAnimation(
        target,
        {
            // Note: this keyframe refers to a keyframe defined in customAnimation.css.
            // If it's not defined in CSS, the animation won't work.
            keyframe: "custom-opacity-in",
            property: "opacity",
            delay: 0,
            duration: 500,
            timing: "linear",
            from: 0,
            to: 1
        }
    );
}

function runCustomShowTransition() {
    var showTransition = WinJS.UI.executeTransition(
        target,
        {
            property: "opacity",
            delay: 0,
            duration: 500,
            timing: "linear",
            to: 1
        }
    );
}
```

If you want to combine multiple animations (as many of the WinJS animations do), note that both of these functions return promises so that you can combine multiple results with `WinJS.Promise.join` to then have a single completed handler in which to do post-processing. This is exactly what WinJS does internally.

And if you know anything about CSS animations and transitions already, you can probably tell that the objects you pass to `executeAnimation` and `executeTransition` are simply shorthand expressions of the CSS styles you would use otherwise. In short, these methods give you an easy way to set up your own custom animations and transitions through the capabilities of CSS. Let's now look at those capabilities directly.

CSS Animations and Transitions

As noted before, many animation needs can be achieved through CSS rather than with JavaScript code running on intervals or animation frames. The WinJS Animations Library, as we've just seen, is entirely built on CSS. Using CSS relieves us from writing a bunch of code that worries about how much to move every element in every frame based on elapsed time and synchronized to the refresh rate. Instead, we can simply declare what we want to happen (perhaps using the `WinJS.UI.executeAnimation` and `WinJS.UI.executeTransition` helpers) and let the host take care of the details. Delegation at its best! In this section, then, let's take a closer look at the capabilities of CSS for WinRT apps.

Another huge benefit of performing animations and transitions through CSS—specifically those that affect only transforms and opacity properties—is that they can be used to create what are called *independent animations* that run on a GPU thread rather than the UI thread. This makes them smoother and more power-efficient than *dependent animations* that are using the UI thread—which is what happens when you create animations in JavaScript using intervals, use CSS animations and transitions with properties other than transform and opacity, or run animations on elements that are partly or wholly obscured by other elements.

We'll come back to this subject in a bit when we look at sample code. As I assume that you're already at least a little familiar with the subject, let's first review how CSS animations and transitions work. I say animations and transitions both because there are, in fact, two separate CSS specifications: CSS animations (<http://www.w3.org/TR/css3-animations/>), and CSS transitions (<http://www.w3.org/TR/css3-transitions/>). So what's the difference?

Normally when a CSS property changes, its value jumps immediately from the old value to the new value, resulting in a sudden visual change. *Transitions* instruct the app host how to change one or more property values gradually, according to specific delay, duration, and timing curve parameters. All of this is declared within a specific style rule for an element (as well as `:before` and `:after` pseudo-elements) using four individual styles:

- `transition-property` (`transitionProperty` in JavaScript) Identifies the CSS properties affected by the transition (the transitionable properties are listed in section 7

of the spec).

- **transition-duration** (**transitionDuration** in JavaScript) Defines the duration of the transition in seconds (fractional seconds are supported, as in `.125s`; negative values are normalized to `0s`).
- **transition-delay** (**transitionDelay** in JavaScript) Defines the delayed start of the transitions relative to the moment the property is changed, in seconds. If a negative value is given, the transition will appear to have started earlier but the effect will not have been visible.
- **transition-timing-function** (**transitionTimingFunction** in JavaScript) Defines how the property values change over time. The functions are `ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`, `cubic-bezier`, `step-start`, and `step-end`. The W3C spec has some helpful diagrams that explain these, but the best way to see the difference is to try them out in running code.

When defining transitions for multiple properties, each value in each style is separated by a comma. For example, a transition for a single property appears as so:

```
#div1 {  
  transition-property: left;  
  transition-duration: 2s;  
  transition-delay: .5s;  
  transition-timing-function: linear;  
}
```

For multiple properties:

```
.class2 {  
  transition-property: opacity, left;  
  transition-duration: 1s, 0.25s;  
}
```

Again, transitions don't specify any actual beginning or ending property values—they define how the change actually happens *whenever* a new property is set through another CSS rule or through JavaScript. So, in the first case above, if `left` is initially 100px and it's set to 300px through a `:hover` rule, it will transition after 0.5 seconds from 100px to 300px over a period of 2 seconds. Doing the math, the visual movement with a `linear` timing function will run at 100px/second. Other timing functions will show different rates of movement at different points along the 2-second duration.

If a bit of JavaScript then sets the value to -200px—ideally after the first transition completes and fires its `transitionend` event—the value will again transition over the same amount of time but now from 300px to -200px (a total of 500px). As a result, the element will move at a higher speed (250px/second, again with the `linear` timing function) because it has more ground to cover for the same transition duration.

What's also true for transitions is that if you assign a style (e.g., `class2` above) to an element,

nothing will happen until an affected property changes value. Changing a style like this also has no effect if a transition is already in progress. The exception is if you change the `transition-property`, in which case that transition will stop. With this, it's important to note that the default value of this property is `all`, so clearing it (setting it to `""`) doesn't stop all transitions...it enables them! You instead need to set the property to `none`.

Note Elements with `display: none` do not run CSS animations and transitions at all, for obvious reasons! The same cannot be said about elements with `display: hidden`, `visibility: hidden`, `visibility: collapsed`, or `opacity: 0`, which means that hiding elements with some means other than `display: none` might end up running animations on nonvisible elements, which is a complete waste of resources. In short, use `display: none`!

Animations work in an opposite manner to transitions. Animations are defined separately from any CSS style rules but are then attached to rules. Assigning that style to an element then triggers the animation immediately. Furthermore, groups of affected properties are defined together in *keyframes* and are thus animated together.

A CSS animation, in other words, is the progressive updating of one or more CSS property values over a period of time. The values change from an initial state to a final state through various intermediate states defined by a set of keyframes. Here's an example (from Scenario 1 of the [HTML Independent Animations sample](#) we'll be referring to):

```
@keyframes move {  
  from { transform: translateX(0px); }  
  50% { transform: translateX(400px); }  
  to { transform: translateX(800px); }  
}
```

More generally:

- Start with `@keyframes <identifier>` where `<identifier>` is whatever name you want to assign to the keyframe (like *move* above). You'll refer to this elsewhere in style rules.
- Within this keyframe, you create any number of *rule sets*, each of which represents a different snapshot of the animated element at different stages in the overall animation, demarked by percentages. The `from` and `to` keywords, as shown above, are simply aliases for 0% and 100%, respectively, and are not necessary since you can just use the straight percentages.
- Within each rule set you then define the desired value of *any number of style properties* (just `transform` in the example above), with each separated by a semicolon as with CSS styles. If a value for a property is the same as in the previous rule set, no animation will occur for that property. If the value is different, the rendering engine will animate the change between the two values of that property across the amount of time equivalent to `<overall animation time> * (<toPercentage> - <fromPercentage>)`. A timing function can also be specified for each rule set using the `animation-timing-function`

style. For example:

```
50% { transform: translateX(400px); animation-timing-function: ease-in;}
```

One thing you'll notice here is that while the keyframe can indicate a timing function, it doesn't say anything about actual *timings*. This is left for the specific style rules that refer to the keyframe. In Scenario 1 of the sample, for instance:

```
.ball {  
  animation-name: move;  
  animation-duration: 2s;  
  animation-timing-function: linear;  
  animation-delay: 0s;  
  animation-iteration-count: infinite;  
  animation-play-state: running;  
}
```

Here, the `animation-name` style (`animationName` in JavaScript) identifies the keyframe to apply. The other `animation-*` styles then describe how the keyframe should execute:

- `animation-duration` (`animationDuration` in JavaScript) The duration of the animation in seconds (fractions allowed, of course). Negatives are the same as `0s`.
- `animation-timing-function` (`animationTimingFunction` in JavaScript) Defines, as with transitions, how the property values are interpolated over time—`ease` (the default), `linear`, `ease-in`, `ease-out`, `ease-in-out`, `cubic-bezier`, `step-start`, and `step-end`.
- `animation-delay` (`animationDelay` in JavaScript) Defines the number of seconds after which the animation will start when the style is applied. This can be negative, as with transitions, which will start the animation partway through its cycle.
- `animation-iteration-count` (`animationIterationCount` in JavaScript) Indicates how many times the animation will repeat (default is 1). This can be a number or `infinite`, as shown above.
- `animation-direction` (`animationDirection` in JavaScript) Indicates whether the animation should play `normal` (forward), `reverse`, `alternate` (back and forth), or `alternate-reverse` (back and forth starting with `reverse`). The default is `normal`.
- `animation-play-state` (`animationPlayState` in JavaScript) Allows you to play or pause an animation. The default state of `running` plays the animation; setting this to `paused` will pause it until you set the style back to `running`.
- `animation-fill-mode` (`animationFillMode` in JavaScript) Defines which property values of the named keyframe will be applied when the animation is not executing, such as during the initial delay or after it is completed. The default value of `none` applies the values of the `0%` or `from` rule set if the direction is `forward` and `alternate`

directions; it applies those of the `100%` or `to` rule set if the direction is reverse or `alternate-reverse`. A fill mode of `backwards` flips this around. A fill mode of `forwards` always applies the `100%` or `to` values (unless the iteration count is zero, in which case it acts like `backwards`). The other option, `both`, is the same as indicating both `forwards` and `backwards`.

- `animation` (`animation` in JavaScript) The shorthand style for all of the above (except for play-state) in the order of name, duration, timing function, delay, iteration count, direction, and fill mode.

Applying a style that contains `animation-name` will trigger the animation for that element. This can happen automatically if the animation is named in a style that's applied by default. It can also happen on demand if the style is assigned to an element in JavaScript or if you set the `animation` property for an element.

Keyframes, while typically defined in CSS, can also be defined in JavaScript. The first step is to build up a string that matches what you'd write in CSS, and then you insert that string to the stylesheet. This is shown in Scenario 7 of the HTML Independent Animations sample (`js/scenario7.js`):

```
var styleSheet = document.styleSheets[1];
var element1 = document.getElementById("ballcontainer");
var animationString = '@keyframes bounce1 {'
    // ...
    + '}';

styleSheet.insertRule(animationString, 0);

window.setImmediate(function () {
    element1.style.animationName = 'bounce1';
});
```

Note how this code uses `setImmediate` to yield to the UI thread before setting the `animationName` property that will trigger the animation. This makes sure that other code that follows (not shown here) will execute first, as it does some other work we want to complete before the animation begins.

More generally, it's good to know that CSS animations and transitions start only when you return from whatever function is setting them up. That is, nothing happens visually until you yield back to the UI thread and the rendering engine kicks in again, just like when you change nonanimated properties. This means you can set up however many animations and transitions as desired, and they'll all execute simultaneously. Using a callback with `setImmediate`, as shown above, is a simple way to say, "Run this code as soon as there is no pending work on the UI thread."⁵⁶ Such a pattern is typically for triggering one or more animations once everything else is set up.

As a final note for this section, you might be interested in [The Guide to CSS Animation: Principles and Examples](#) from Smashing Magazine. This will tell you a lot about animation design beyond just

⁵⁶ For more on this topic, see <http://dvcs.w3.org/hg/webperf/raw-file/tip/specs/setImmediate/Overview.html>.

how CSS animations are set up in your code.

The Independent Animations Sample

Turning now to the [HTML Independent Animations sample](#), Scenario 1 gives a demonstration of an independent versus a dependent animation by eating some time on the UI thread (that is, blocking that thread) according to a slider. As a result, the top red ball (see image below) moves choppily, especially as you increase the work on the UI thread by moving the slider. The green ball on the bottom, on the other hand, continues to move smoothly the whole time.



What's tricky to understand about this sample is that both balls use the same *ball* CSS style rule we saw in the previous section. In fact, just about everything about the two elements is exactly the same. So why does the movement of the red ball get choppy when additional work is happening on the UI thread?

The secret is in the `z-index: -1;` style on the red ball in `css/scenario1.css` (and a corresponding lack of `position: static` which negates `z-index`). For animations to run independently, they must be free of obstruction. This really gets into the subject of how layout is being composed within the HTML/CSS rendering engine of the app host, as an animating element that's somewhere in the middle of the z-order might end up being independent or dependent. The short of it is that the `z-index` style is the only lever that's available for you to pull here.

As I noted before, independent animations are limited to those that affect only the transform and opacity properties for an element. If you animate any property that affects layout, like `width` or `left`, the animation will run as dependent (and similar results can be achieved with a scaling and translation transform anyway). Other factors also affect independent animations, as described on the [Animating](#) topic in the documentation. For example, if the system lacks a GPU, if you overload the GPU with too many independent animations, or if the elements are too large, some of the animations will revert to dependent. This is another good reason to be purposeful in your use of animations—overusing them will produce a terrible user experience on lower-end devices, thereby defeating the whole point of using animations to enhance the user experience.

The other scenarios of the HTML Independent Animations sample lets you play with CSS transitions and animations by setting values within various controls and then running the animation. Scenarios 2

and 3 work with CSS transitions for 2D and 3D transforms, respectively, with an effect of the latter shown in Figure 11-2. As you can see, the element that the sample animates is the container for all the input controls! Scenarios 5 and 6 then let you do similar things with CSS animations. In all these cases, the necessary styles are set directly in JavaScript rather than using declarative CSS, so look in the .js files and not the .css files for the details.

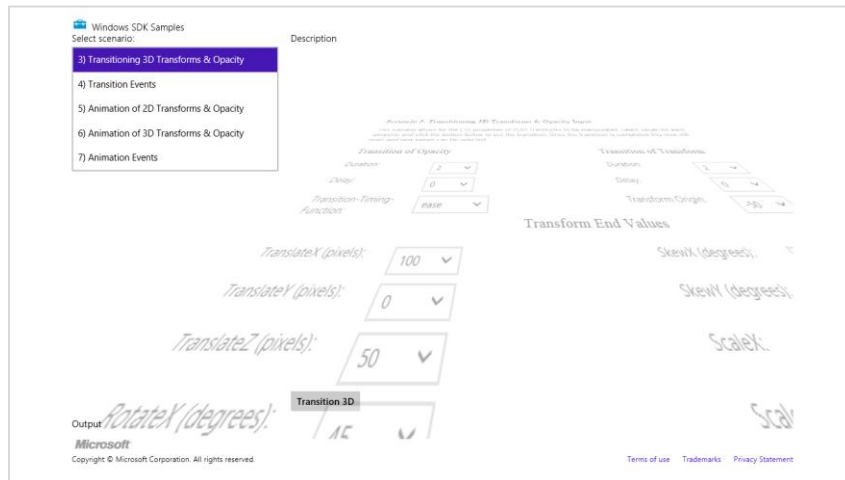


FIGURE 11-2 Output of Scenario 3 of the HTML Independent Animations sample.

Scenarios 4 and 7 then show something we haven't talked about yet, which are the few simple events that are raised for transitions and animations (and actually have nothing to do with independent versus dependent animations). In the former case, any element on which you execute a CSS transition will fire `transitionstart` and `transitionend` events. You can use these to chain transitions together.

With animations, there are three events: `animationstart` (which comes after any delay has passed), `animationend` (when the animation finished), and `animationiteration` (at the end of each iteration, unless `animationend` also fires are the same time). As with transitions, all of these can be used to chain animations or otherwise synchronize them. The `animationiteration` event is also helpful if you need to run a little code every time an animation finishes a cycle. In such a handler you might check conditions that would cause you to stop an animation, in which case you can set the `animationPlayState` to paused when needed.

Rolling Your Own: Tips and Tricks

If you're anything like me, I imagine that one of the first things you did when you started playing with JavaScript is to do some kind of animation: set up some initial conditions, create an timer with `setInterval`, do some calculations in the handler and update elements (or draw on a canvas), and

keep looping until you're done. After all, this sort of thing is at the heart of many of our favorite games! (For an introductory discussion on this, just in case you haven't done this on your own yet, see [How to Animate Canvas Graphics](#).)

As such, there is considerable wisdom available in the community on this subject if you decide to go this route. I put it this way because by now, having looked at the WinJS animations library and the capabilities of CSS, you should be in a good position to decide whether you actually *need* to go this route at all! Some people have estimated that a vast majority of animations needed by most apps can be handled entirely through CSS: just set a style and let the host do the rest. But if you still need to do low-level animation, the first thing you should do is ask an important question:

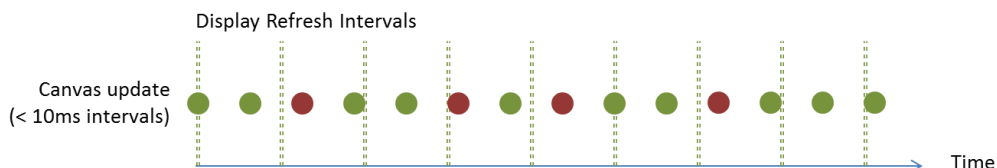
What is the appropriate animation interval?

This is a very important question because oftentimes developers have no idea what kind of interval to use for animation. It's not so much of an issue for long intervals, like 500ms or 1s, but developers often just use 10ms because it seems "fast enough."

To be honest, 10ms is overkill for a number of reasons. 60 frames per second (fps)—an animation interval of 16.7ms—is about the best that human beings can even discern and is also the best that most displays can even handle in the first place. In fact, the best results are obtained when your animation frames are synchronized with the screen refresh rate.

Let's explore this a little more. Have you ever looked at a screen while eating something really crunchy, and noticed how the pixels seem to dance all over the place? This is because display devices aren't typically just passive viewports onto the graphics memory. Instead, displays (even LCD and LED displays) typically cycle through graphics memory at a set refresh rate, which is most commonly 60Hz or 60fps (but can also be 50Hz or 100Hz).

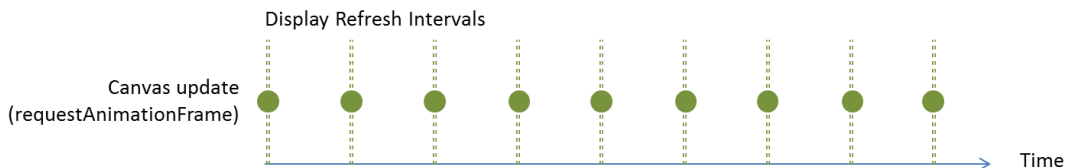
This means that trying to draw animations at an interval faster than the refresh rate is a waste of time, is a waste of power (it has been shown to reduce battery life by as much as 25%!), and results in dropped frames. The latter point is illustrated below, where the red dots are frames that get drawn on something like a [canvas](#) but never make it to the screen because another frame is drawn before the screen refreshes:



This is why it's common to animate on multiples of 16.7ms using [setInterval](#). However, using 16.7 assumes a 60Hz display refresh, which isn't always the case. The right solution, then, for WinRT apps in JavaScript and web apps both, is to use [requestAnimationFrame](#). This API simply takes a function to call for each frame:

```
requestAnimationFrame(renderLoop);
```

You'll notice that there's not an interval parameter; the function rather gives you a way to align your frame updates with display refreshes so that you draw only when the system is ready to display something:



What's more, `requestAnimationFrame` also takes page visibility into account, meaning that if you're not visible (and animations are thus an utter waste), you won't be asked to render the frame at all. This way you don't need to handle page visibility events yourself to turn animations on and off: you can just rely on the behavior of `requestAnimationFrame` directly.

Tip If you really want an optimized display, consider doing all drawing work of your app (not just animations) within a `requestAnimationFrame` callback. That is, when processing a change, as in response to an input event, update your data and call `requestAnimationFrame` with your rendering function rather than doing the rendering immediately.

Calling this method once will invoke your callback for a single frame. To keep up a continuous animation, your handler should call `requestAnimationFrame` again. This is shown in the [Using requestAnimationFrame for power efficient animations sample](#) (this wins second place for long sample names!), which draws and animates a clock with a second hand:



The first call to `requestAnimationFrame` happens in the page's `ready` method, and then the callback refreshes the request (`js/scenario1.js`):

```
window.requestAnimationFrame(renderLoopRAF);

function renderLoopRAF() {
  drawClock();
  window.requestAnimationFrame(renderLoopRAF);
}
```

where the `drawClock` function gets the current time and calculates the angle at which to draw the

clock hands:

```
function drawClock() {  
  // ...  
  
  // Note: this is modified from the sample to only create a Date once, not each time  
  var date = new Date();  
  var hour = date.getHours();  
  var minute = date.getMinutes();  
  var second = date.getSeconds();  
  
  // ...  
  
  var sDegree = second / 60 * 360 - 180;  
  var mDegree = minute / 60 * 360 - 180;  
  var hDegree = ((hour + (minute / 60)) / 12) * 360 - 180;  
  
  // Code to use the context's translate, rotate, and drawImage methods to render each clock hand  
}
```

Here's a challenge for you: *What's wrong with this code?* Run the sample and look at the second hand. Then think about how `requestAnimationFrame` aligns to screen refresh cycles with an interval like 16.7ms. What's wrong with this picture?

What's wrong is that even though the second hand is moving visibly only *once per second*, the `drawClock` code is actually executing nearly *50, 60, or 100 times more frequently* than that! Thus the "Efficient and Smooth Animations" title that the sample shows on screen is anything but! Indeed, if you run Task Manager, you can see that this simple "efficient" clock is ironically consuming 15–20% of the CPU. Yikes!

 Efficient Animations JS sample	20.2%	39.6 MB	0 MB/s	0 Mbps
--	-------	---------	--------	--------

Remember that an interval aligned with ~16.7ms screen refreshes (on a 60Hz display) implies 60fps rendering; if you don't need that much, you should skip frames yourself (thereby saving processing power) according to elapsed time and not blindly redraw (as this sample is doing). In fact, if all we need is a once-per-second movement in a clock like this, repeated calls to `requestAnimationFrame` is sheer overkill. We could instead use `setInterval(function () { requestAnimationFrame(drawClock) }, 1000)` and be done (that is, coordinating the 1s interval with a screen refresh). If you make this change in the `ready` method, for example, the CPU usage will drop precipitously:

 Efficient Animations JS sample	0.5%	39.6 MB	0 MB/s	0 Mbps
--	------	---------	--------	--------

But let's say we actually want to put 60fps animation and 20% of the CPU to good use—in that case, we should at least make the clock's second hand move smoothly, which can be done by simply adding milliseconds into the angle calculation:

```
var second = date.getSeconds() + date.getMilliseconds() / 1000;
```

Still, 20% is a lot of CPU power to spend on something so simple and 60fps is still serious overkill.

~10fps is probably sufficient for good effect. In this case we can calculate elapsed time within `renderLoopRAF` to call `drawClock` only once 0.1 seconds have passed:

```
var lastTime = 0;

function renderLoopRAF() {
  var fps = 10; // Target frames per second
  var interval = 1000 / fps;
  var curTime = Math.floor(Date.now() / interval);

  if (lastTime !== curTime) {
    lastTime = curTime;
    drawClock();
  }

  requestAnimationFrame(renderLoopRAF);
}
```

That's not quite as smooth—10fps creates the sense of a slight mechanical movement—but it certainly has much less impact on the CPU:

 Efficient Animations JS sample	5.2%	39.8 MB	0 MB/s	0 Mbps
--	------	---------	--------	--------

I encourage you to play around with variations on this theme to see what kind of interval you can actually discern with your eyes. 10fps and 15fps gives a sense of mechanical movement; at 20fps I don't see much difference from 60fps at all, and the CPU is running at about 7–10%. You might also try something like 4fps (quarter-second intervals) to see the effect. In this chapter's companion content I've included a variation of the original sample where you can select from a various of target rendering rates.

The other thing you can do in the modified sample is draw the hour and minute hand at fractional angles. In the original code, the minute hand will move suddenly when the second hand comes around to the top. Many analog clocks don't actually work this way: their complex gearing moves both the hour and the minute hand ever so slightly with every tick. To simulate that same behavior, we just need to include the seconds in the minutes calculation, and the resulting minutes in the hours, like so:

```
var second = date.getSeconds() + date.getMilliseconds() / 1000;
var minute = date.getMinutes() + second / 60;
var hour = date.getHours() + minute / 60;
```

In real practice, like a game, you'd generally want to avoid just running a continuous animation loop like this: if there's nothing moving on the screen that needs animating (for which you might be using `setInterval` just as a timer) and there are no input events to respond to, there's no reason to call `requestAnimationFrame`. Also, be sure when the app is paused that you stop calling `requestAnimationFrame` until the animation starts up again. (You can also use `cancelAnimationFrame` to stop one you've already requested.) The same is true for `setTimeout` and `setInterval`: don't generate unnecessary calls to your callback functions unless you really need to do the animation. For this, use the `visibilitychanged` event to know if your app is visible on screen. While `requestAnimationFrame` takes visibility into account (the sample's CPU use will drop to 0% before it is

suspended), you need to do this on your own with `setTimeout` and `setInterval`.

In the end, the whole point here is that really understanding the animation interval you need (that is, your frame rate) will help you make the best use of `requestAnimationFrame`, if that's needed, or `setInterval`/`setTimeout`. They all have their valid uses to deliver the right user experience with the right level of consumption of system resources.

What We've Just Learned

- In the desktop control panel, users can elect to disable most (that is, nonessential) animations. Apps should honor this, as does WinJS, by checking the `Windows.UI.ViewManagement.UISettings.animationsEnabled` property.
- The WinJS animations library has many built-in animations that embody the Windows 8 personality. These are highly recommended for apps to use for the scenarios they support, such as content and page transitions, selections, list manipulation, and others.
- All WinJS animations are built using CSS and thus benefit from hardware acceleration. When the right conditions are met, such animations run in the GPU and are thus not affected by activity on the UI thread.
- Apps can also use CSS animations and transitions directly, according to the W3C specifications.
- Apart from WinJS and CSS, apps can also use functions like `setInterval` and `requestAnimationFrame` to implement direct frame-by-frame animation. The `requestAnimationFrame` method is best to align frames with the display refresh rate, leading to the best overall performance.

Chapter 12

Contracts

Recently I discovered a delightfully quirky comedy called *Interstate 60* that is full of delightfully quirky characters. One of them, played by Chris Cooper, is a former advertising executive who, having discovered he was terminally ill with lung cancer, decided to make up for a career built on lies by encouraging others to be more truthful. As such, he was very particular about agreements and contracts, especially those in writing.

We really get to see the quirkiness of the character in a scene at a gas station where he's approached by a beggar with a sign, "Will work for food." Seeing this, he offers the man an apple in exchange for cleaning his car's windshield. But when the man refuses to honor the written contract on his sign, Cooper's character gets increasingly upset over the breach...to the point where he announces his terminal illness, rips open his shirt, and reveals the dynamite wrapped around his body and the 10-second timer that's already counting down!

In the end, he drives away with a clean windshield and the satisfaction of having helped someone—in his delightfully quirky way—to fulfill their part of a written contract. And he reappears later in the movie in a town that's 100% populated with lawyers; I'll leave it to you to imagine the result, or at least enjoy the film.

Agreements between two parties are exceptionally important in a civil society—dynamite aside!—as they are in a well-running computer system. Agreements are especially important where apps provide extensions to the system and where apps written by different people at different points in time cooperate between themselves to fulfill a certain task.

Such is the nature of various contracts within Windows 8, which as a whole is perhaps one of the most powerful features of the entire system. With any given contract, one party is the consumer or receiver of information that's managed through the contract. The other party is then the provider of that information. The contract itself is then generic: neither party needs any specific knowledge of the other, just knowledge of their side of the contract. It might not sound like much, but what this allows is a degree of extensibility that gets richer and richer as more apps that support those contracts are added to the system. I figure that when users really start to experience what these contracts provide, they'll more and more look for and choose apps from the Windows Store that use contracts to create powerful user experiences.

Within the apps themselves, consuming contracts typically happens through an API call, such as the file pickers, or is already built into the system through UI like the Charms bar. *Providing* information for a contract is often the more interesting part, because an app needs to announce the capability through its manifest and then handle different forms of activation. Again, we've seen this already with Settings, and now we can explore many of the others.

The table below summarizes all the contracts and other extensions in Windows 8 (in alphabetical order), some of which serve to allow apps to work together while others serve to allow apps to extend system functionality. Full descriptions can be found on [App contracts and extensions](#). Those that are covered in this chapter are colored in green: share, search, file type and protocol associations, file pickers, cached file updater, and contacts. Others contracts have been or will be covered in the chapters indicated, and a few I've simply left for you to explore through the linked documentation and samples.

Tip For a comparison of the different options for exchanging data—the share contract, the clipboard, and the file save picker contract—see [Sharing and exchanging data](#) on the Windows Developer Center. It outlines different scenarios for each option and when you might implement more than one in the same app.

Contract/Extension	Provider	Consumer	Description, Documentation, and Samples
Account picture provider (Chapter 14)	Apps that can take a picture	Windows (account picture)	When user changes an account picture, they can either select an existing one or take a new one; see Account picture name sample .
AutoPlay (Chapter 15)	Apps that want to be listed as AutoPlay option	Windows	See Auto-launching with AutoPlay and the Removable Storage sample .
Background tasks (Chapters 13 and 14)	Apps that have background tasks	Windows	Allows apps to run small tasks in the background (that is, when otherwise suspended) without user interaction. See Introduction to background tasks , as well as Chapter 13. Background file transfers are a special case supported by specific APIs; see Transferring data in the background and Chapter 14.
Cached file updater	Apps that provide access to their data through file pickers and want to synchronize updates	Apps using the file picker API and the file APIs to manage them.	Provider apps can allow the consumer to maintain a cached copy of a file. Through this contract, the provider can manage updates between the local copy and the source copy. See Integrating with file picker contracts .
Camera settings (Chapter 15)	Apps with custom camera UI	Windows Camera Capture UI	See Developing Windows 8 device apps for cameras .
Contact picker	Apps that manage contact data (like an address book)	Apps that use the contact picker API (like email)	Launches an app to provide a list of possible contacts to select. See Managing user contacts .
File activation (file type association)	Apps that can open files of a particular type	Windows Explorer and apps that use the launcher API	Launches an app to open/service a file when needed. See How to handle file activation and Auto-launching with file and URI associations .
File open picker/file save picker	App with data that can appear as files to other apps for opening and/or saving (there are two separate contracts).	Apps using the file picker API (also certain Windows features)	Makes data that is otherwise hidden inside and managed by apps appear as if they were part of the file system. See Integrating with file picker contracts .
Game explorer	Game apps with a Game Definition File	Windows (parental controls)	Manages age ratings for games. See Creating a GDF file .
Play To (Chapter 10)	Apps that can play media to a DLNA device	Windows (Devices charm > Connect)	See Streaming media to devices using Play To .
Print task settings (Chapter 15)	Printer device apps	Windows (Device charm > Print)	See Developing Windows 8 device apps for printers .
Protocol activation	Apps that can open URIs that	Windows Explorer and apps	Launches an app to open/service a URI when

(URI scheme association)	begin with a particular URI scheme	that use the launcher API	needed. See How to handle protocol activation and Auto-launching with file and URI associations .
Search	Apps with searchable data	Windows (Search charm)	Provides the ubiquitous ability to search any app from anywhere. See Adding search to an app .
Settings (Chapter 8)	Apps with settings	Windows (Settings charm)	Provides a standard place for app settings. See Adding app settings .
Share	Apps with sharable data	Apps that can receive data to incorporate into itself or share to a	Provides a linkage of data transfer between apps so that source apps don't need to be particularly aware of individual targets like FaceBook, etc. See Adding share .
SSL/certificates (Chapter 14)	Apps that need to install a certificate	Apps needing to supply a certificate to another service	See Encrypting data and working with certificates .

Share

Though Search appears at the top of the Charms bar, the first contract I want to look at in depth is Share—after all, it's one of the first things you learn as a child! In truth, I'm starting with Share because we've already seen the source side of the story starting back in Chapter 2, "Quickstart," with the Here My Am! app, and our coverage here will also include a brief look at the age-old clipboard at the end of this section.

Here's what we've already learned about Share, with the more complete process illustrated in Figure 12-1:

- An app with sharable content listens for the `datarequested` event from the object returned from `Windows.ApplicationModel.DataTransfer.DataTransferManager.getForCurrentView()`. This event is fired whenever the user invokes the Share charm. Note that if an app doesn't listen for this event at all, the Share charm will show a default "unable to share" message (one that is certain to be disappointing to users!).
- In its event handler, the app determines whether it has anything to share in its current state. If it does, it populates the `Windows.ApplicationModel.DataTransfer.DataPackage` provided in the event.
- Based on the data formats in the package, Windows—that is, an agent called the *share broker* who manages the contract—determines the share target apps to display to the user. The user can also control which apps are shown through Change PC Settings > Share.
- When the user picks a target, the associated app is activated and receives the data package to process however it wants.

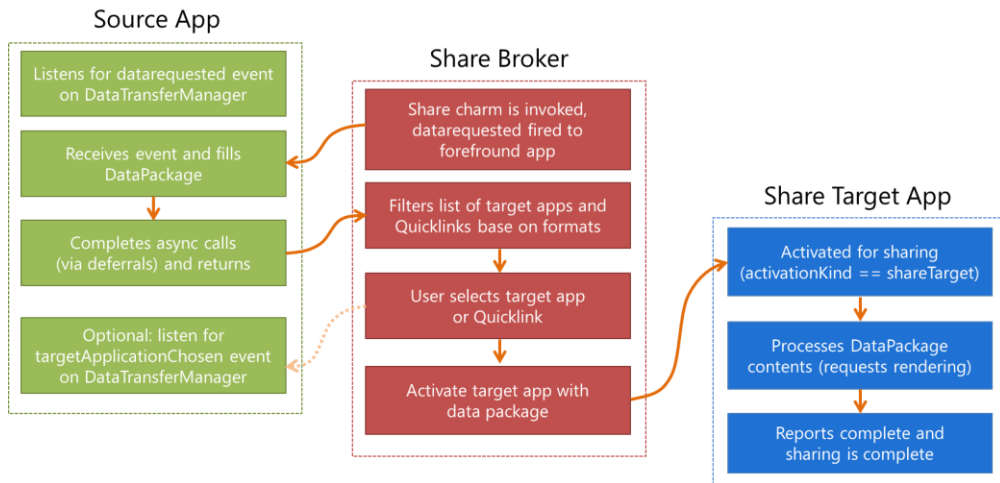


FIGURE 12-1 Processing the Share contract as initiated by the user's selection of the Share charm.

This whole process provides a very convenient shortcut for users to take something they love in one app and get it into another app with a simple edge gesture and picking an app. What's very cool about the Share contract is that the source doesn't have to care what happens to the data—its only role is to provide whatever data is appropriate for sharing at the moment the user invokes the Share charm (if, in fact, there is appropriate data—sometimes there isn't). This liberates source apps from the burden of having to predict, anticipate, or second-guess what users might want to do with the data. Perhaps they want to email it, share it via social networking, drop it into a content management app...who knows?

Well, only the user knows, so what the share broker does with that data is let the user decide! Given the data package from the source, the broker matches the formats in that package to apps that know how to handle them and displays that list to the user. That list can contain apps, for one, but also something called a quicklink (a `Windows.ApplicationModel.DataTransfer.ShareTarget.Quicklink` object, to be precise), which is serviced by some app but is much more specific. For instance, when an email app is shown as an option for sharing, the best it can do is create a new message with no particular recipients. A quicklink, however, can identify specific email addresses, say, for a person or persons you email frequently. The quicklink, then, is essentially an app plus specific configuration information.

Whatever the case, some app is launched when the user selects a target. With the Share contract, the app is launched with an `activationKind` of `shareTarget`. This tells it not to bring up its default UI but to rather show a specific share pane (with light-dismiss behavior) in which the user can refine exactly what is being shared and how. A share target for a social network, for instance, will often provide a place to add a comment on the shared data before posting it to the social network. An email app would provide a means to edit the message before sending it. A front-end app for a photo service could allow for adding a caption, specifying a location, identifying people, and so on. You get the idea. All of this combines together to provide a smooth flow from having something to share to an app that

facilitates the sharing.

Overall, then, the Share contract gets apps connected to one another for this common purpose without either one having to know anything about the other. This creates a very extensible and scalable experience: since all the potential target choices appear only in the Share charm pane, they never need to clutter a source app as we see happening on many web pages. (This is the “content before chrome” design principle in action.) Source apps also don’t need to update themselves when a new target becomes popular (e.g., a new kind of social network): all that’s needed is a single target app. As for those target apps, they don’t have to evangelize themselves to the world: through the contract, source apps are automatically able to use any target apps that come along in the future. And from the end user’s point of view, their experience of the Share charm gets better and better as they acquire more Share-capable apps.

At the same time, it is possible for the source app to know something about how its shared data is being used. Alongside the `datarequested` event, the `DataTransferManager` also fires a `targetApplicationChosen` event to those sources who are listening. The `eventArgs` in this case contain only a single property: `applicationName`. This isn’t really useful for any other WinRT APIs, mind you, but is something you can tally within your own analytics. Such data can help you understand whether you’d provide a better user experience by sharing richer data formats, for example, or, if you see that certain target apps also support particular custom formats, you can start supporting those in a future update.

Source Apps

Let’s complete our understanding of source apps now by looking at a number of details we haven’t fully explored yet, primarily around how the source populates the data package and the options it has for handling the request. For this purpose, I suggest you obtain and run both the [Sharing content source app sample](#) and the [Sharing content target app sample](#). We’ll be looking at both of these, and the latter provides a helpful way to see how a target app will consume the data package created in the source.

The source app sample provides a number of scenarios that demonstrate how to share different types of data. They also show how to programmatically invoke the Share charm with the following line of code (if it really fits in your app scenario, because doing so is generally not recommended):

```
Windows.ApplicationModel.DataTransfer.DataTransferManager.ShowShareUI();
```

Doing so will, as when the user invokes the charm, trigger the `datarequested` event where `eventArgs.request` object is a [Windows.ApplicationModel.DataTransfer.DataRequest](#) object. This request object contains two properties and two methods:

- `data` is the `DataPackage` to populate. It contains methods to set make various data formats available, though it’s important to note that not all formats will be immediately rendered. Instead, they’re rendered only when a share target asks for them.

- `deadline` is a `Date` property indicating the time in the future when the data you're making available will no longer be valid (that is, will not render). This recognizes that there might be an indeterminate amount of time between when the source app is asked for data and when the target actually tries to use it. With delayed rendering, as noted above for the `data` property, it's possible that some transient source data might disappear after some time. By indicating that time in `deadline`, rendering requests that occur past the deadline will be ignored.
- `failWithDisplayText` is a method to tell the share broker that sharing isn't possible right now, along with a string that will tell the user why (perhaps the lack of a usable selection). You call this when you don't have appropriate data formats or an appropriate selection to share, or if there's an error in populating the data package for whatever reason. The text you provide will then be displayed in the Share charm (and thus should be localized). Scenario 8 of the source app sample shows the use of this in the simple case when you have no sharable data when Share is invoked.
- `getDeferral` provides for async operations you might need to perform while populating the data package (just like other deferrals elsewhere in the WinRT API), and should also be used if you need more than 200ms in the handler (its timeout period). That is, once you return from the `datarequested` event, the share broker normally assumes that your data package is ready. If you need to wait for an async operation to complete, on the other hand, you can call `getDeferral` to obtain a [DataRequest-Deferral](#) object, whose `complete` method you call within the completed handler of the async operation.

The basic structure of a `datarequested` handler, then, will attempt to populate the minimal properties of `eventArgs.request.data` and call `eventArgs.request.failWithDisplayText` when errors occur. We see this structure in most of the scenarios in the sample:

```
var dataTransferManager =
    Windows.ApplicationModel.DataTransfer.DataTransferManager.getForCurrentView();
dataTransferManager.addEventListener("datarequested", dataRequested);

function dataRequested(e) {
    var request = e.request;

    // Title is required
    var dataPackageTitle = document.getElementById("titleInputBox").value;

    if ( /* Check if there is appropriate data to share */ ) {
        request.data.properties.title = dataPackageTitle;

        // The description is optional.
        var dataPackageDescription = document.getElementById("descriptionInputBox").value;
        request.data.properties.description = dataPackageDescription;

        // Call request.data.setText, setUri, setBitmap, setData, etc.
    } else {
```

```

        request.failWithDisplayText(/* Error message */ );
    }
}

```

As we see here, the `data.properties` object (of type [DataPackagePropertySet](#)) is where you set things like a title and description for the data package. Other properties are [applicationListingUri](#) (the URI of your app's page in the Windows Store, which should be set to the return value of [Windows.ApplicationModel.Store.CurrentApp.linkUri](#)⁵⁷), [applicationName](#) (a string, which helps share targets gather the same kind of information that the source can obtain from the [targetApplicationChosen](#) event), [fileTypes](#) (a string vector, where strings should come from the [StandardDataFormats](#) enumeration but can also be custom formats), [size](#) (the number of items when the data in the package is a collection [e.g., files]), and [thumbnail](#) (a stream containing a thumbnail image; obtaining this image is typically why you'd use the [DataRequest.getDeferral](#) method). Beyond this the `data.properties` object also supports custom properties through its [insert](#), [remove](#), and other methods. This makes it possible for the source app to pass custom properties along with custom formats, making all of this extensible if new data formats are widely adopted in the future.

Within this code structure above, the primary job of the source app is to populate the data package by calling the package's various `set*` methods. For standard formats, which are again those described in the [StandardDataFormats](#) enumeration, there are discrete methods: [setText](#), [setUri](#), [setHtmlFormat](#), [setRtf](#) (rich text format, a comparably ancient precursor to HTML), [setBitmap](#), and [setStorageItems](#) (for files and folders). All of these except for [setRtf](#) are represented in the source app sample as follows:

Sharing text—Scenario 1 (js/text.js):

```

var dataPackageText = document.getElementById("textInputBox").value;
request.data.setText(dataPackageText);

```

Sharing a link—Scenario 2 (js/link.js), which can be used for local and remote content alike:

```

request.data.setUri(new Windows.Foundation.Uri(document.getElementById("linkInputBox").value));

```

Sharing an image and a storage item—Scenario 3 (js/image.js):

```

var imageFile; // A StorageFile obtained through the file picker

// In the data requested event
var streamReference = Windows.Storage.Streams.RandomAccessStreamReference.createFromFile(imageFile);
request.data.properties.thumbnail = streamReference;

// It's recommended to always use both setBitmap and setStorageItems for sharing a single image
// since the Target app may only support one or the other

```

⁵⁷ This URI along with the `applicationName` would allow a target to indicate where the data originally came from, especially for scenarios where the data goes to a social network, in an email message, or elsewhere off the device with the source app. This way, recipients can be invited to acquire the source app themselves, so source apps will probably want to include it. You always want to use the [Windows.ApplicationModel.Store.CurrentApp.linkUri](#) property to populate this field because you won't know your URI until your completed app is uploaded to the Store the first time.


```
// Put the image file in an array and pass it to setStorageItems
request.data.setStorageItems([imageFile]);
```

```
// The setBitmap method requires a RandomAccessStreamReference
request.data.setBitmap(streamReference);
```

Sharing files—Scenario 4 (js/file.js):

```
var selectedFiles; // A collection of StorageFile objects obtained through the file picker
```

```
// In the data requested event
request.data.setStorageItems(selectedFiles);
```

As for sharing HTML, this can be quite simple if you just have HTML in a string:

```
request.data.setHtmlFormat(someHtml);
```

For this purpose you might find the

[Windows.ApplicationModel.DataTransfer.HtmlFormatHelper](#) object, well, helpful, as it provides methods to help build properly formatted markup. What's also true with HTML is that it often refers to other content like images that aren't directly contained in the markup. So how do you handle that? Fortunately, the designers of this API thought through the scenario: you employ the data package's [resourceMap](#) property to associate relative URLs in the HTML with an image stream. We see this in Scenario 6 of the sample (hs/html.js):

```
var path = document.getElementById("htmlFragmentImage").getAttribute("src");
var imageUri = new Windows.Foundation.Uri(path);
var streamReference = Windows.Storage.Streams.RandomAccessStreamReference.createFromUri(imageUri);
request.data.resourceMap[path] = streamReference;
```

The other interesting part of Scenario 6 is that it actually replaces the data package in the [eventArgs](#) with a new one that is created as follows:

```
var range = document.createRange();
range.selectNode(document.getElementById("htmlFragment"));
request.data = MSApp.createDataPackage(range);
```

As you can see, the [MSApp.createDataPackage](#) method takes a DOM range (in this case a portion of the current page) and creates a data package from it, where the package's [setHtmlFormat](#) method is called in the process (which is why you don't see that method called explicitly in Scenario 6). For what it's worth, there is also [MSApp.createDataPackageFromSelection](#) that does the same job with whatever is currently selected in the DOM. You would obviously use this if you have editable elements on your page from which you'd like to share.

Sharing Multiple Data Formats

As shown in Scenario 3, it is certainly allowable—and *encouraged!*—to share data in as many formats as makes sense, thereby enabling more potential targets. All this means is that you call the various [set*](#) methods for each format within your [datarequested](#) handler, which also includes calling [setData](#) for

custom formats and `setDataProvider` for deferred rendering, as described in the next two sections.

Custom Data Formats: schema.org

Long ago, I imagine, API designers decided it was an exercise in futility to try to predict every data format that apps might want to exchange in the future. The WinRT API is no different, so alongside the format-specific `set*` methods of the `DataPackage` we find the generic `setData` method. This takes a format identifier (a string) and the data to share as is illustrated in Scenario 7 of the sample using the format `"http://schema.org/Book"` and data in a JSON string:

```
request.data.setData(dataFormat, JSON.stringify(book));
```

Since the custom format identifier is just a string, you can literally use anything you want here; a very specific format string might be useful, for example, in a sharing scenario where you want to target a very specific app, perhaps one that you authored yourself. However, unless you're very good at evangelizing your custom formats to the rest of the developer community (and want to budget for such!), chances are that other share targets won't have any clue what you're talking about.

Fortunately, there is a growing body of conventions for custom data formats maintained by <http://schema.org>. This site is, in other words, the point of agreement where custom formats are concerned, so we highly recommend that you draw formats from here. See <http://schema.org/docs/schemas.html> for a complete list.

You can also use these custom formats alongside standard formats. Take a look at the JSON book data used in the sample:

```
var book = {
  type: "http://schema.org/Book",
  properties: {
    image: "http://sourceuri.com/catcher-in-the-rye-book-cover.jpg",
    name: "The Catcher in the Rye",
    bookFormat: "http://schema.org/Paperback",
    author: "http://sourceuri.com/author/jd_salinger.html",
    numberOfPages: 224,
    publisher: "Little, Brown, and Company",
    datePublished: "1991-05-01",
    inLanguage: "English",
    isbn: "0316769487"
  }
};
```

You can easily express this same data as plain text, as HTML (or RTF), as a link (perhaps to a page with this information), and an image (of the book cover).

Deferrals and Delayed Rendering

Deferrals, as mentioned before are a simple mechanism to delay completion of the `datarequested` event until the deferral's `complete` method is called. The documentation for [DataRequest.getDeferral](#) shows an example of using this when loading an image file:

```

var deferral = request.getDeferral();

Windows.ApplicationModel.Package.current.installedLocation.getFileAsync("images\\smalllogo.png")
    .then(function (thumbnailFile) {
        request.data.properties.thumbnail =
            Windows.Storage.Streams.RandomAccessStreamReference.createFromFile(thumbnailFile);
        return Windows.ApplicationModel.Package.current.installedLocation.getFileAsync(
            "images\\logo.png");
    })
    .done(function (imageFile) {
        request.data.setBitmap(
            Windows.Storage.Streams.RandomAccessStreamReference.createFromFile(imageFile));
        deferral.complete();
    });

```

Delayed rendering is a different matter, though the process typically employs the deferral. The purpose here is to avoid rendering the shared data until a target actually requires it, sometimes referred to as a *pull operation*. That is, all the other `set*` methods we've seen copy the full data into the package—with delayed rendering, you instead call the data package's `setDataProvider` method with a data format identifier and a function to call when the data is needed. Here's how it's done in Scenario 5 of the source app sample where `imageFile` is selected with a file picker:

```

// When sharing an image, don't forget to set the thumbnail for the DataPackage
var streamReference = Windows.Storage.Streams.RandomAccessStreamReference.createFromFile(imageFile);
request.data.properties.thumbnail = streamReference;
request.data.setDataProvider(Windows.ApplicationModel.DataTransfer.StandardDataFormats.bitmap,
    onDeferredImageRequested);

```

As indicated here, it's a really good idea to provide a thumbnail with delayed rendering, so the target app has something to show the user. Then, when the target needs the full data, the data provider function gets called—in this case, `onDeferredImageRequested`:

```

function onDeferredImageRequested(request) {
    if (imageFile) {
        // Here we provide updated Bitmap data using delayed rendering
        var deferral = request.getDeferral();

        var imageDecoder, inMemoryStream;

        imageFile.openAsync(Windows.Storage.FileAccessMode.read).then(function (stream) {
            // Decode the image
            return Windows.Graphics.Imaging.BitmapDecoder.createAsync(stream);
        }).then(function (decoder) {
            // Re-encode the image at 50% width and height
            inMemoryStream = new Windows.Storage.Streams.InMemoryRandomAccessStream();
            imageDecoder = decoder;
            return Windows.Graphics.Imaging.BitmapEncoder.createForTranscodingAsync(
                inMemoryStream, decoder);
        }).then(function (encoder) {
            encoder.bitmapTransform.scaledWidth = imageDecoder.orientedPixelWidth * 0.5;
            encoder.bitmapTransform.scaledHeight = imageDecoder.orientedPixelHeight * 0.5;
            return encoder.flushAsync();
        }).done(function () {

```

```

        var streamReference = Windows.Storage.Streams.RandomAccessStreamReference
            .createFromStream(inMemoryStream);
        request.setData(streamReference);
        deferral.complete();
    }, function (e) {
        // didn't succeed, but we still need to release the deferral to avoid
        // a hang in the target app
        deferral.complete();
    });
}
}

```

Note that this function receives a simplified hybrid of the [DataRequest](#) and [DataPackage](#) objects: a [DataProviderRequest](#) that contains [deadline](#) and [formatId](#) properties, a [getDeferral](#) method, and a [setData](#) method through which you provide the data that matched [formatId](#). The [deadline](#) property, as you can guess, is the same as what the [datarequested](#) handler might have stored in the [DataRequest](#) object.

Target Apps

Looking back to Figure 12-1, we can see that while the interaction between a source app and the share broker is driven by the single [datarequested](#) event, the interaction between the broker and a target app is a little more involved. For one, the broker needs to determine which apps can potentially handle a particular data package, for which purpose each target app includes appropriate details in its manifest. When an app is selected, it gets launched with an [activationKind](#) of [shareTarget](#), in response to which it should show a specific share UI rather than the full app experience.

Let's see how all this works with the [Sharing content target app sample](#) whose appearance is shown in Figure 12-2 (borrowing from Figure 2-22 we saw ages ago). Be sure to directly run this app once in Visual Studio so that it's effectively installed and it will appear on the list of apps when we invoke the Share charm. We'll also be using the Share Contract Target item template in the tools.

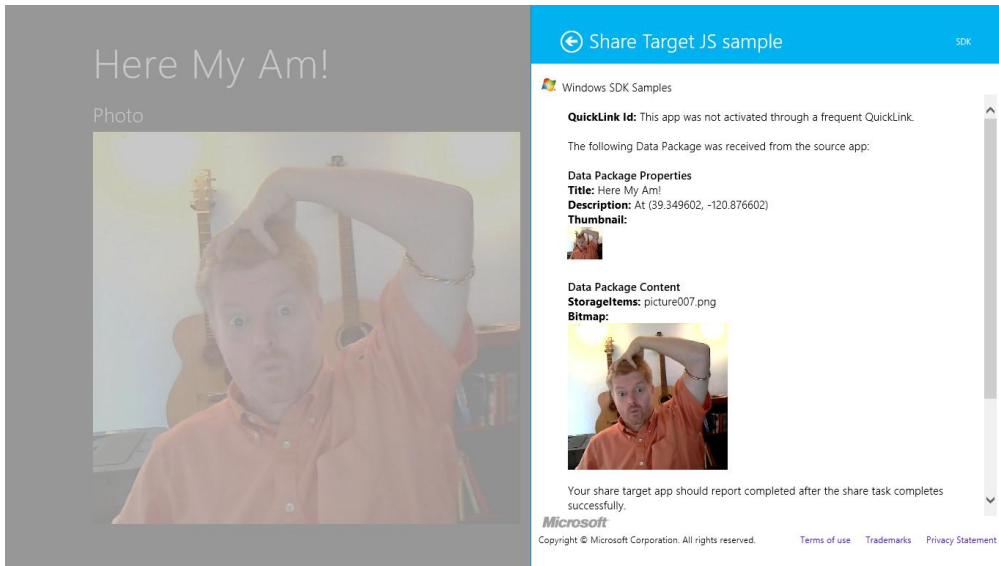


FIGURE 12-2 The appearance of the Share Target app sample (the right-hand nonfaded part).

The first step for a share target is to declare the data formats it can accept in the Declarations section of the app manifest, along with the page that will be invoked when the app is selected as a target. As shown in Figure 12-3, the target app sample declares it can handle text, URI, bitmap, html, and the <http://schema.org/Book> formats, and it also declares it can handle whatever files might be in a data package (you can also be specific here). Way down at the bottom it then points to target.html as its Share target page.

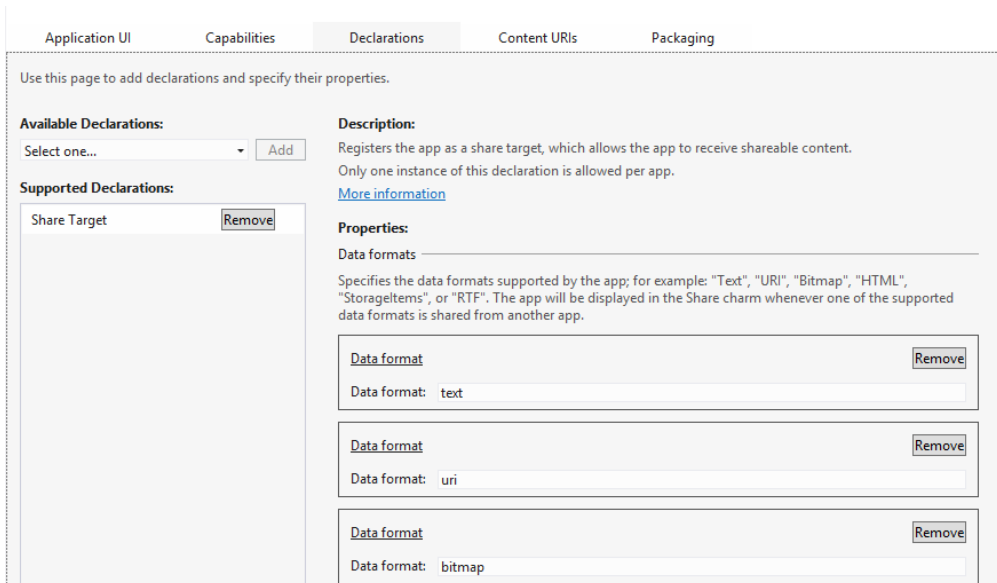


FIGURE 12-3 The Share target app sample's manifest declarations.

The Share start page, `target.html`, is just a typical HTML page with whatever layout you require for performing the share task. This page typically operates independently of your main app: when your app is chosen through Share, this page is loaded and activated by itself and thus has an entirely separate script context. This page should not provide navigation to other parts of the app and should thus load only whatever code is necessary for the sharing task. (The Executable and Entry Point options are not used for apps written in HTML and JavaScript; those exist for apps written in other languages.)

Much of this structure is built for you automatically through the Share Target Contract *item* template provided by Visual Studio and Blend, as shown in Figure 12-4; the dialog appears when you right-click your project and select Add > New Item, or select the Project > Add New Item... menu command.

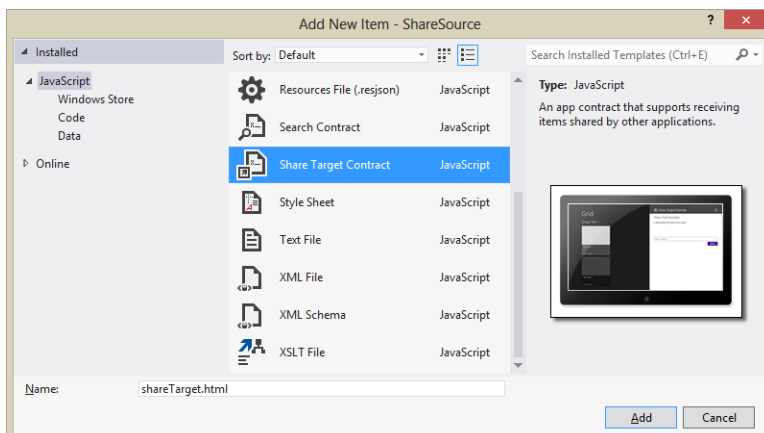


FIGURE 12-4 The Share Target Contract item template in Visual Studio and Blend.

This item template will give you HTML, JS, and CSS files for the share target page and will add that page to your manifest declarations along with text and URI formats—so you'll want to update those as needed.

Before we jump into the code, a few notes about the design of a share target page, summarized from [Guidelines for sharing content](#):

- Maintain the app's identity and its look and feel, consistent with the primary app experience.
- Keep interactions simple to quickly complete the share flow: avoid text formatting, tagging, and setup tasks, but do consider providing editing capabilities especially if posting to social networks or sending in a message. (See Figure 12-5 from the Mail app for an example.) A social networking target app would generally want to include the ability to add comments as well; a photo-sharing target would probably include the ability to add captions.
- Avoid navigation: sharing is a specific task flow, so use inline controls and errors instead of switching to other pages. Another reason to avoid this is that the share page of the target app typically runs in its own script context, so being able to navigate elsewhere in the app within a separate context could be very confusing to users.
- Avoid links that would distract from or take the user away from the sharing experience. Remember that sharing is a way to shortcut the oft-tedious process of getting data from one app to another, so keep the target app focused on that purpose.
- Avoid light-dismiss flyouts since the Share charm already works that way.
- Acknowledge user actions when you start sending the data off (to an online service, for example) so that users know something is actually happening.
- Put important buttons within reach of the thumbs on a touch device; refer to [Windows 8 Touch Posture](#) topic in the documentation for placement guidance.
- Make previews match the actual content—in other words, don't play tricks on the user!

With this page design, it's good to know that you do *not* need to worry about different view states—this page really just has one state. It does need to adapt itself well to varying dimensions, mind you, but not different view states. Basing the layout on a CSS grid with fractional rows and columns is a good approach here.

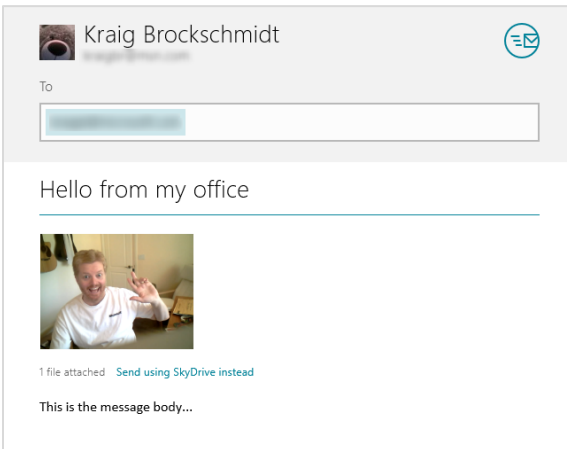


FIGURE 12-5 The sharing UI of the Windows Mail app (the bottom blank portion has been cropped); this UI allows editing of the recipient, subject, and message body, and allows for sending an image as an attachment or as a link to SkyDrive.

Caution Because a target app can receive data from any source app, it should treat all such content as untrusted and potentially malicious, especially with HTML, URIs, and files. As a result, the target app should avoid adding such HTML or file contents to the DOM, executing code from URIs, navigating to the URI or some other page based on the URI, modifying database records, using `eval` with the data, and so on.

Let's now look at the contents of the template's JavaScript file as a whole, because it shows us the basics of being a target. First, as you can see, we have the same structure as a typical `default.js` for the app, using the `WinJS.Application` object's methods and events.

```
(function () {
    "use strict";

    var app = WinJS.Application;
    var share;

    function onShareSubmit() {
        document.querySelector(".progressindicators").style.visibility = "visible";
        document.querySelector(".commentbox").disabled = true;
        document.querySelector(".submitbutton").disabled = true;

        // TODO: Do something with the shared data stored in the 'share' var.

        share.reportCompleted();
    }

    // This function responds to all application activations.
    app.onactivated = function (args) {
        var thumbnail;

        if (args.detail.kind === Windows.ApplicationModel.Activation.ActivationKind.shareTarget) {
```



```

document.querySelector(".submitButton").onclick = onShareSubmit;
share = args.detail.shareOperation;

document.querySelector(".shared-title").textContent = share.data.properties.title;
document.querySelector(".shared-description").textContent =
    share.data.properties.description;

thumbnail = share.data.properties.thumbnail;
if (thumbnail) {
    // If the share data includes a thumbnail, display it.
    args.setPromise(thumbnail.openReadAsync().then(function displayThumbnail(stream) {
        document.querySelector(".shared-thumbnail").src =
            window.URL.createObjectURL(stream, { oneTimeOnly: true }));
    }));
} else {
    // If no thumbnail is present, expand the description and
    // title elements to fill the unused space.
    document.querySelector("section[role=main] header").style
        .setProperty("-ms-grid-columns", "0px 0px 1fr");
    document.querySelector(".shared-thumbnail").style.visibility = "hidden";
}
}
};

app.start();
})();

```

When this page is loaded and activated (during which time the app's splash screen will appear), its `WinJS.Application.onactivated` event will fire—again independently of your app's main `activated` handler that's typically in `default.js`. As a share target you just want to make sure that the `activationKind` is `shareTarget`, after which your primary responsibility is to provide a preview of the data you'll be sharing along with whatever UI you have to edit it, comment on it, and so forth. Typically, you'll also have a button to complete or submit the sharing, during which you tell the share broker that you've completed the process.

The key here is the `args.detail.shareOperation` object provided to the `activated` handler. This is a `Windows.ApplicationModel.DataTransfer.ShareTarget.ShareOperation` object, whose `data` property contains a read-only package called a `DataPackageView` from which you obtain all the goods:

- To check whether the package formats you can consume, use the `contains` method or the `availableFormats` collection.
- To obtain data from the package, use its `get*` methods such as `getTextAsync`, `getBitmapAsync`, and `getDataAsync` (for custom formats). When pasting HTML you can also use the `getResourceMapAsync` method to get relative resource URIs. The view's `properties` like the `thumbnail` are also useful to provide a preview of the data.

As you can see, the Share target item template code above doesn't do anything with shared data other than display the title, description, and thumbnail; clearly your app will do something more by

requesting data from the package, like the examples we see in the Share target app sample. Its `target.js` file contains an `activated` handler for the `target.html` page, and it also displays the thumbnail in the data package by default. It then looks for different data formats and displays those contents if they exist:

```
if (shareOperation.data.contains(Windows.ApplicationModel.DataTransfer.StandardDataFormats.text)) {
    shareOperation.data.getTextAsync().done(function (text) {
        displayContent("Text: ", text, false);
    });
}
```

The same kind of code appears for the simpler formats. Consuming a bitmap is a little more work but straightforward:

```
if (shareOperation.data.contains(Windows.ApplicationModel.DataTransfer.StandardDataFormats.bitmap)) {
    shareOperation.data.getBitmapAsync().done(function (bitmapStreamReference) {
        bitmapStreamReference.openReadAsync().done(function (bitmapStream) {
            if (bitmapStream) {
                var blob = MSApp.createBlobFromRandomAccessStream(bitmapStream.contentType,
                    bitmapStream);
                document.getElementById("imageHolder").src = URL.createObjectURL(blob,
                    { oneTimeOnly: true });
                document.getElementById("imageArea").className = "unhidden";
            }
        });
    });
}
```

For HTML, it looks through the markup for `img` elements and then sets up their `src` attributes from the resource map (the `iframe` already has the HTML content from the package by this time):

```
var images = iframe.contentDocument.documentElement.getElementsByTagName("img");
if (images.length > 0) {
    shareOperation.data.getResourceMapAsync().done(function (resourceMap) {
        if (resourceMap.size > 0) {
            for (var i = 0, len = images.length; i < len; i++) {
                var streamReference = resourceMap[images[i].getAttribute("src")];
                if (streamReference) {
                    // Call a helper function to map the image element's src
                    // to a corresponding blob URL generated from the streamReference
                    setResourceMapURL(streamReference, images[i]);
                }
            }
        }
    });
}
```

The `setResourceMapURL` helper function does pretty much what the bitmap-specific code did, which is call `openReadAsync` on the stream, call `MSApp.createBlobFromRandomAccessStream`, pass that blob to `URL.createObjectURL`, and set the `img.src` with the result.

After the target app has completed a sharing operation, it should call the `ShareOperation.report-`

`Completed` method, as shown earlier with the template code. This lets the system know that the data package has been consumed, the share flow is complete, and all related resources can be released. The Share target sample does this when you explicitly click a button for this purpose, but normally you should call the method whenever you've completed the share. Do be aware that calling `reportCompleted` will close the target app's sharing UI, so avoid calling it as soon as the target activates: you want the user to feel confident that the operation was carried out.

Long-Running Operations

When you run the Share target sample and invoke the Share charm from a suitable source app, there's a little expansion control near the bottom labeled "Long-running Share support." If you expand that, you'll see some additional controls and a bunch of descriptive text, as shown in Figure 12-6. The buttons shown here tie into a number of other methods on the `ShareOperation` object alongside `reportCompleted` that help Windows understand exactly how the share operation is happening within the target: `reportStarted`, `reportDataRetrieved`, `reportSubmittedBackgroundTask`, and `reportError`. As you can see from the descriptions in Figure 12-6, these generally relate to telling Windows when the target app has finished cooking its meal and the system can clean the dishes and put away the utensils, so to speak:

- `reportStarted` informs Windows that your sharing operation might take a while, as if you're uploading the data from the package to another place, or just sending an email attachment with what ends up being large images and such. This specific indicates that you're obtained user input such that the share pane can be dismissed.
- `reportDataRetrieved` informs Windows that you've extracted what you need from the data package such that it can be released. If you've called `MSApp.createBlobFromRandomAccessStream` for an image stream, for example, the blob now contains a copy of the image that's local to the target app. If you're using images from the package's `resourceMap`, on the other hand, you'll not want to call `reportDataRetrieved` unless you explicitly make a copy of those references whose URIs refer to bits inside the data package. In any case, if you need to hold onto the package throughout the operation, you don't need to call this method as you'll later call `reportCompleted` to release the package.
- `reportSubmittedBackgroundTask` tells Windows that you've started a background transfer using the [Windows.Networking.BackgroundTransfer.BackgroundUploader](#) class. As the sample description in Figure 12-6 indicates, this lets Windows know that it can suspend the target app and not disturb the sharing operation. If you call this method with a local copy of the data being uploaded, go ahead and call `reportCompleted` method so that Windows can clean up the package; otherwise wait until the transfer is complete.
- `reportError` lets Windows know if there's been an error during the sharing operation.

Report Completed

Long-running Share Support ^

This API is required if your app supports uploading a format that may take some time, such as images or videos. A user should be able to dismiss your app and have the upload continue in the background while they do other things. In order for the dismiss behavior to work correctly, you need to report to the share platform that you finished getting user input. After you call this, a user can go back to the share pane and see your application in the share progress list.

Report Started

This API is optional and helps Windows to optimize resource usage of the system. You should report this if you have finished extracting data from the Data Package so that Windows can suspend or terminate the source app as necessary to reclaim system resources.

Report Data Retrieved

This API is optional and helps Windows to optimize resource usage of the system. You should report this if you have called the Windows Runtime Background Transfer class to upload your content. Then Windows can suspend your app as necessary to reclaim system resources. If you use this API, call it after Report Started.

Report Submitted To BackgroundTask

If for any reason the long-running share was unsuccessful and failed in the background, you should report failure and include a message for the user about how they can recover from the error. When the user goes back to the share pane they can see your message in the progress list. You must never call Report Error if your app is visible in the foreground.

Error message:

Report Error

FIGURE 12-6 Expanded controls in the Share target sample for Long-Running Share Support. The Report Completed button is always shown and isn't specific to long-running tasks despite its placement in the sample's UI. Don't let that confuse you!

Quicklinks

The last aspect of the Share contract for us to explore is something we mentioned early on in this section: quicklinks. Just like doing online check-in before an airplane flight, the purpose of these is to streamline the Share process such that users don't need to re-enter information in a target app. For example, if a user commonly shares data with particular people through email, each contact can be a quicklink for the email app. If a user commonly shares with different people or groups through a social networking app, those people and/or groups can be represented with quicklinks. And as these targets are much more user-specific than target apps in general, the Share charm UI shows these at the top of its list (see Figure 12-7 below).

Each quicklink is associated with and serviced by a particular target app and simply provides an identifier to that target. When the target is invoked through a quicklink, it then uses that identifier to retrieve whatever data is associated with that quicklink and prepopulate or otherwise configure its UI. It's important to understand that quicklinks contain *only* an identifier, so the target app must store and

retrieve the associated data from some other source, typically local app data where the identifier is a filename, the name of a settings container, or so forth. The target app could also use roaming app data or the cloud for this purpose, but quicklinks themselves do not roam to another device—they are strictly local. Thus, it makes the most sense to store the associated data locally as well.

A quicklink itself is just an instance of the [Windows.ApplicationModel.DataTransfer.Quicklink](#) class. You create one with the `new` operator and then populate its `title`, `thumbnail`, `supportDataFormats`, `supportedFileTypes`, and `id` properties. The data formats and file types are what Windows uses to determine if this quicklink should be shown in the list of targets for whatever data is being shared from a source app (independent of the app's manifest declarations). The `title` and `thumbnail` are used to display that choice in the Share charm, and the `id` is what gets passed to the target app when the quicklink is chosen.

Tip For the thumbnail, use an image that's more specifically representative of the quicklink (such as a contact photo) rather than just the target app. This helps distinguish the quicklink from the general use of the target app.

An app then registers a quicklink with the system by passing it to the [ShareOperation.reportCompleted](#) method. As this is the *only* way in which a quicklink is registered, it really tells us that creating a quicklink *always* happens as part of another sharing operation. It's a way to create a specific target that might save the user time and encourage them to choose your target app again in the future.


Let's follow the process within the Share target app sample to see how this all works. First, when you invoke the Share charm and choose the sample, you'll see that it provides a check box for creating a quicklink, as shown in Figure 12-7. When you check this, it provides fields in which you can enter an id and a title (the thumbnail just uses a default image). When you press the Report Completed button, it calls [reportCompleted](#) and the quicklink is registered. On subsequent invocations of the Share charm with the appropriate data formats from the source app, this quicklink will then appear in the list, as shown in Figure 12-8 where the app that services the quicklink is always indicated under the provided title.

When reporting completed, you can optionally add a QuickLink to make it easier for users to repeat the way they share most often. This saves them from having to select that person or group in your app every time they share to them.

☒ Add a QuickLink (optional)

QuickLink Id:

Title: ✕

Icon: 

Long-running Share Support ▼

Report Completed

FIGURE 12-7 Controls to create a quicklink in the Share target app sample.

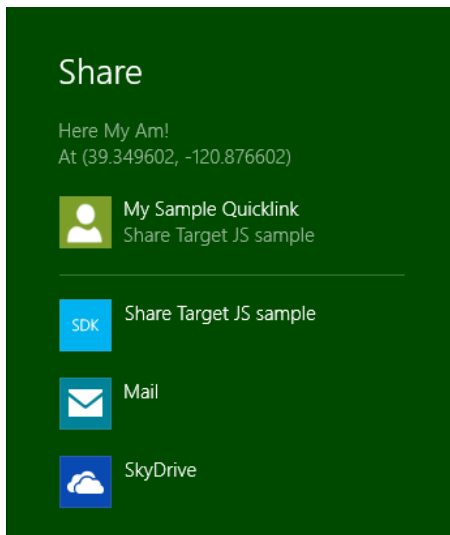


FIGURE 12-8 A quicklink from the Share target app sample as it appears in the Share charm target list.

Here's how the Share target app sample creates the quicklink within the function `reportCompleted` (`js/target.js`) that's called when you click the Report Completed button (some error checking omitted):

```
if (addQuickLink) {
    var quickLink = new Windows.ApplicationModel.DataTransfer.ShareTarget.QuickLink();

    var quickLinkId = document.getElementById("quickLinkId").value;
    quickLink.id = quickLinkId;

    var quickLinkTitle = document.getElementById("quickLinkTitle").value;
    quickLink.title = quickLinkTitle;

    // For quicklinks, the supported FileTypes and DataFormats are set independently
```

```
// from the manifest
var dataFormats = Windows.ApplicationModel.DataTransfer.StandardDataFormats;
quickLink.supportedFileTypes.replaceAll(["*"]);
quickLink.supportedDataFormats.replaceAll([dataFormats.text, dataFormats.uri, dataFormats.bitmap,
    dataFormats.storageItems, dataFormats.html, customFormatName]);

// Prepare the icon for a QuickLink
Windows.ApplicationModel.Package.current.installedLocation.getFileAsync("images\\user.png")
    .done(function (iconFile) {
        quickLink.thumbnail = Windows.Storage.Streams.RandomAccessStreamReference
            .createFromFile(iconFile);
        shareOperation.reportCompleted(quickLink);
    });
```

Again, the process is just to create the `Quicklink` object, set its properties (perhaps settings a more specific thumbnail such as a contact person's picture), and pass it to `reportCompleted`. In the Share target app sample, you can see that it doesn't actually store any other local app data; for its purposes the properties in the quicklink are sufficient. Most target apps, however, will likely save some app data for the quicklink that's associated with the `quicklink.id` property and reload that data when activated later on through the quicklink.

When the app is activated in this way, the `eventArgs.detail.shareOperation` object within the `activated` event handler will contain the `quickLinkId`. The Source target app simply displays this id, but your would certainly use it to load app data and prepopulate your share UI:

```
// If this app was activated via a QuickLink, display the QuickLinkId
if (shareOperation.quickLinkId !== "") {
    document.getElementById("selectedQuickLinkId").innerText = shareOperation.quickLinkId;
    document.getElementById("quickLinkArea").className = "hidden";
}
```

Note the one detail here that when the target app is invoked through a quicklink, it doesn't display the same UI to create a quicklink, because doing so would be redundant. However, if the user edited the information related to the quicklink, you might provide the ability to update the quicklink (meaning to update the data you save related to the id) or to create a new quicklink (with a new id).

The Clipboard

Well before the Share contract was ever conceived, the mechanism we know as the Clipboard was once the poster child of app-to-app cooperation. And while it may not garner any media attention nowadays, it's still a tried-and-true means for apps to share and consume data.

For Windows 8 apps, clipboard interactions build on the same `DataPackage` mechanisms we've already seen for sharing, so everything we've learned about populating that package, using custom formats, and using delayed rendering still apply. Indeed, if you make data available on the clipboard, you should make sure the same data is available for the Share contract!

The question now is how to wire up commands like copy, cut, and paste—from the app bar, a context menu, or keystrokes—should an app provide them for its own content (many controls handle

the clipboard automatically). For this we turn to the [Windows.ApplicationModel.DataTransfer.Clipboard](#) class.

As demonstrated in the [Clipboard app sample](#), the processes here are straightforward. For copy and cut:

- Create a new `Windows.ApplicationModel.DataTransfer.DataPackage` (or use `MSApp.createDataPackage` or `MSApp.createDataPackageFromSelection`), and populate it with the desired data.

```
var dataPackage = new Windows.ApplicationModel.DataTransfer.DataPackage();
dataPackage.setText(textValue);
//...
```

- (Optional) Set the package's `requestedOperation` property to values from [DataPackageOperation](#): `copy`, `move`, `link`, or `none` (the latter is used with delayed rendering). Note that these values can be combined using the bitwise OR operator, as in:

```
var dpo = Windows.ApplicationModel.DataTransfer.DataPackageOperation;
dataPackage.requestedOperation = dpo.copy | dpo.move | dpo.link;
```

- Pass the data package to [Windows.ApplicationModel.DataTransfer.Clipboard.setContent](#):

```
Windows.ApplicationModel.DataTransfer.Clipboard.setContent(dataPackage);
```

To perform a paste:

- Call [Windows.ApplicationModel.DataTransfer.Clipboard.getContent](#) to obtain a read-only data package called a [DataPackageView](#):

```
var dataView = Windows.ApplicationModel.DataTransfer.Clipboard.getContent();
```

- Check whether it contains formats you can consume with the `contains` method (alternately, you can check the contents of the `availableFormats` vector):

```
if (dataView.contains(Windows.ApplicationModel.DataTransfer.StandardDataFormats.text)) {
    //...
}
```

- Obtain data using the view's `get*` methods such as `getTextAsync`, `getBitmapAsync`, and `getDataAsync` (for custom formats). When pasting HTML, you can also use the `getResourceMapAsync` method to get relative resource URIs. The view's `properties` like the `thumbnail` are also useful, along with the `requestedOperation` value or values.

```
dataView.getTextAsync().done(function (text) {
    // Consume the data
})
```

If at any time you want to clear the clipboard contents, call the [Clipboard](#) class's `clear` method.

You can also make sure data is available to other apps even if yours is shut down by calling the `flush` method (which will trigger any deferred rendering you might have set up).

Apps that use the clipboard also need to know when to enable to disable a paste command depending on available formats. At any time you can get the data package view from the clipboard and use its `contains` method or `availableFormats` property and decide accordingly. You should also then listen to the Clipboard object's `contentChanged` event, which will be fired when you or some other app calls its `setContent` method, at which time you'd again enable or disable the commands. Of course, you won't receive this event when your app is suspended, so you should refresh the state of those commands within your resuming handler.

Again, the Clipboard app sample provides example of these various scenarios, including copy/paste of text and HTML (Scenario 1); copy and paste of an image (Scenario 2); copy and paste of files (Scenario 3); and clearing the clipboard, enumerating formats, and handling `contentChanged` (Scenario 4).

Note, finally, that pasted data can come from anywhere. Apps that consume data from the clipboard should, like a share target, treat the content they receive as potentially malicious and take appropriate precautions.

Search

Search has become such a ubiquitous feature for apps that the designers of Windows 8 decided to provide a system-level keyword search UI (with built-in Input Method Editor support) directly alongside Share, Devices, and Settings in the charms bar, as shown in Figure 12-9. This means that apps don't need to (and generally shouldn't) provide their own UI controls to perform a search, and by participating in this contract, the user can not only easily search the app that's in the foreground but also quickly and easily search within other apps without having to go off and start those apps separately. It also means that users never need to explicitly start your app to search within it. Simply by changing the search target within the search pane, that target app is launched and asked to perform a search with the current keywords. This is also what makes Search work even if the current foreground app doesn't support the contract at all—the search target just defaults to the first app in the list.

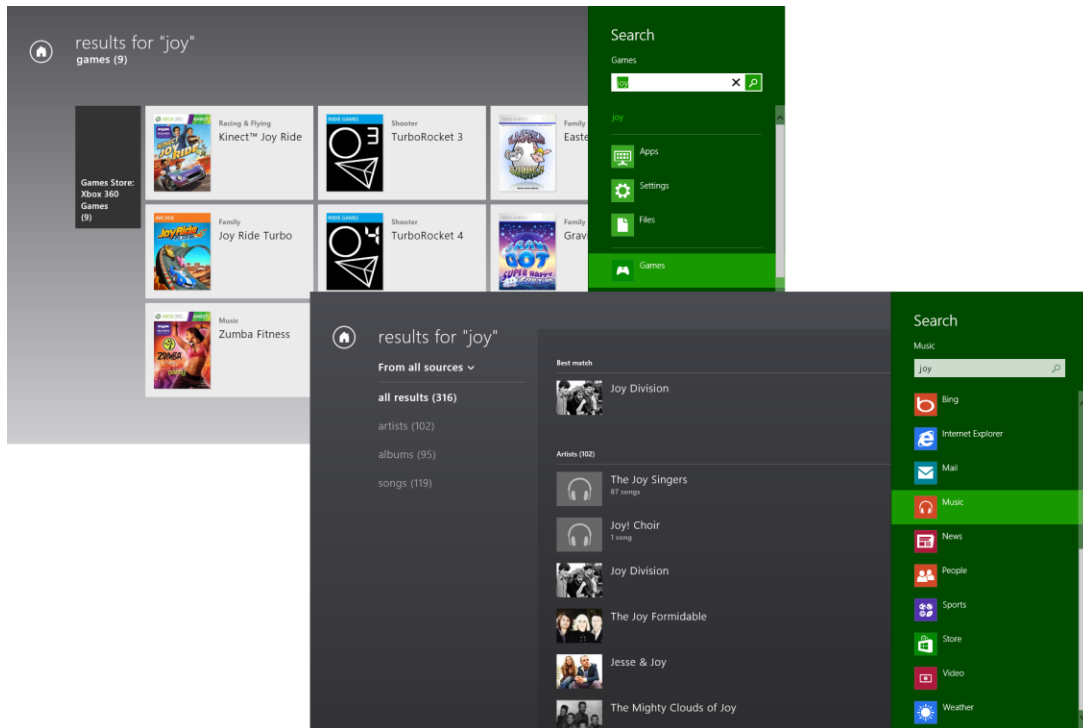


FIGURE 12-9 The Search pane invoked through the Search charm, with results shown in the Games app and the Photos app. As with Share, the user can control which apps are shown through Change PC Settings > Search. That same settings panel also allows the user to clear search history and control a few other aspects of the UI.

The Search contract that makes this happen is composed of a set of interactions between the Windows-provided Search UI and the search target app. (In this section, when I refer to a *target* app, I'm referring now to search, not share.) This interaction communicates the keywords (even if empty) to the app when the user presses Enter, clicks the icon to the right of the entry field, or changes apps. The interaction also allows the target app to provide suggested search terms, as well as suggested results (with result-specific graphics) that appear in the search pane directly, as shown in Figure 12-10.

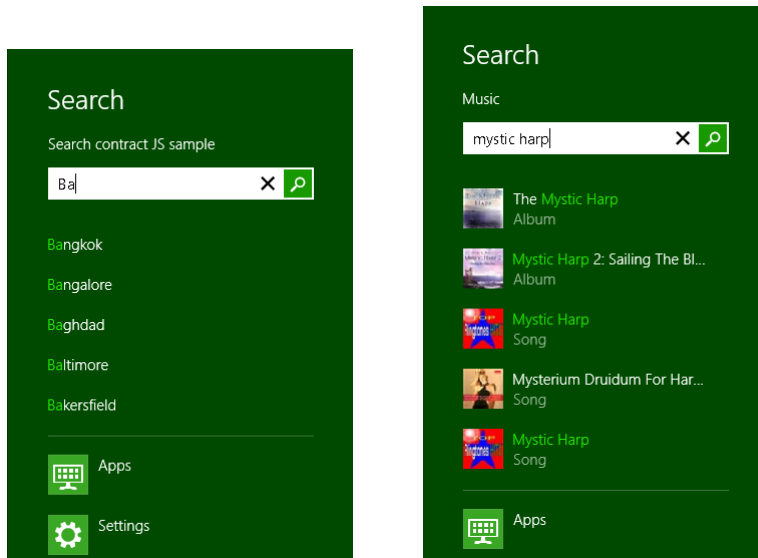


FIGURE 12-10 Suggested searches (left) and search results (right) from a target app appear directly in the search pane.

Designwise, Search should work with whatever data the app manages, whether local or online (or both); it's really the primary means to search within everything that the app can access. For this reason, Microsoft highly recommends that apps *don't* provide their own search UI (which otherwise distracts from the app's content) unless it's really all the app does and where it would need additional search criteria up-front. Otherwise, it's best to let the user first search through the charm and then perhaps filter, sort, and otherwise organize the results within the app through on-canvas or app bar commands. On the flip side, the Search charm is *not* intended for finding data *within* a page; that is, it is expected that apps provide their own controls for essentially scanning and highlighting results that are already in view (like the find function in browsers). Many details on such design questions can be found on [Guidelines for search](#).

Searching within an app effectively navigates the app to its search results page, as we see in Figure 12-9, and thus activates the app in the same script context as when it's run normally. Again, if the app needs to be launched to service the search contract, it will be launched directly into that page (we'll see this mechanism shortly). Tapping on a result then navigates the app directly to the details for that result. Of course, if the app was already running (as is the case when Search was invoked on a running app), the result page's back button should navigate to whatever page the user was on when they invoked Search. Even if the app is launched to service the Search charm, it's helpful to provide the user with a means to navigate to its home page, especially when there are no results from which to navigate elsewhere.

Let's now look at the basic search contract interaction, after which we'll explore the richer aspects of search suggestions, suggested results, and type to search.

Search in the App Manifest and the Search Item Template

An app's life as a search target begins, as with other contracts, in the app manifest on the Declarations tab, as shown in Figure 12-11.

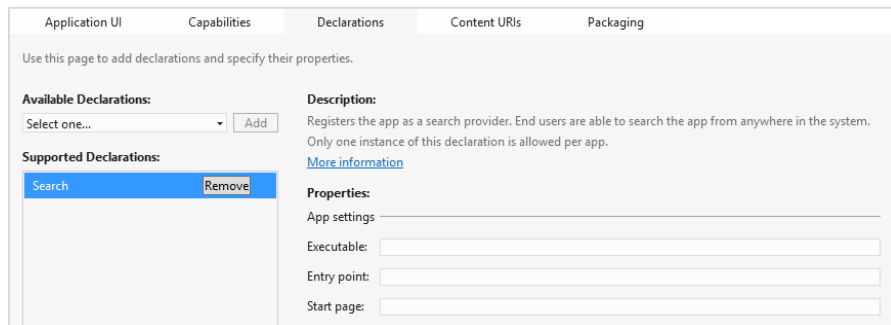


FIGURE 12-11 The Search declarations page within Visual Studio; typically, the App settings properties are left blanks in an HTML/JavaScript app.

Since search is not tied to any particular data format like share, all you really need to specify here is a Start page, if in fact you want it to be separate from the rest of your app at all. Unlike the share contract, search is much more integrated with in-app navigation: when the user taps a result on your results page, you want to navigate to that page directly as if they'd tapped on the same item in some other list. Similarly, if the user taps the back button in your results page, they should navigate to whatever page they were on when the charm was first invoked. For this reason, then, activation via search typically gets handled by through the app's main `activated` event. We'll get to that in the next section.

An easy way to add the Search contract is through the Search contract item template in Visual Studio and Blend. (You can see this listed in Figure 12-4 just above the Share target contract.) If you right-click your project and select Add > New Item, or use the Project > Add New Item... menu command, you can choose the Search Contract item in from the list of templates. This will add the Search declaration in your app manifest and add three page control files (.html, .js. and .css file) for a search results page. There's not much exciting to show here visually because the template code very much relies on there being some real data to work with. Nevertheless, the template gives you a great structure to work from, including the recommended UI for providing filters and so forth. Some further details can be found on [Adding a Search Contract item template](#).

Basic Search and Search Activation

The most basic interaction with the Search contract is receiving a query when the app is already running. This is a great example of how search really just triggers navigation in the app, rather than loading a new page with a separate script context, as happens with share targets. To receive such a query, you need only listen to the `querysubmitted` event of the `Windows.ApplicationModel.Search.SearchPane` object. The exact code looks something like this

where `searchPageURI` identifies the results page:

```
var searchPane = Windows.ApplicationModel.Search.SearchPane.getForCurrentView();
searchPane.onquerysubmitted = function (eventArgs) {
    WinJS.Navigation.navigate(searchPageURI, eventArgs);
};
```

The `eventArgs` object here will be a [SearchPaneQuerySubmittedEventArgs](#) that contains just two properties: `queryText` (the contents of the text box in the search pane) and `language` (the BCP 47 language tag currently in used). In the code above, these are just passed to the `WinJS.Navigation.navigate` method that passes them onto to the results page (whatever `searchPageURI` contains). From there, that page will just process `queryText` appropriate to `language` and populate the page contents with appropriate items. For this purpose an app typically uses a `ListView` control, as you might expect for a variable-length results collection.

Through the same `SearchPane` object you can also set the `placeholderText` property with whatever you'd like to have showing when the Search charm is invoked and the search box is empty. Its `show` method also allows you to show the pane programmatically, its `visible` property and `visibilitychanged` event will tell you its status, and its `queryText` property will give you the current contents of the input control.

You can also listen for its `querychanged` event whose `eventArgs` will contain `queryText` and `language` properties, as with `querysubmitted`, along with a `linguisticDetails` property that provides details about text entered through an Input Method Editor (IME). This event is a precursor to `querySubmitted` and is appropriate if you have logic you need to run outside of providing suggestions, such as previewing results (the behavior you see on the start screen when searching for apps, also known as word wheeling).

We'll return to the other events of the `SearchPane` object in the sections that follow. Before that we want to see how search affects app activation. Again, because a search results page is much more closely integrated with an app's typical operation, you usually leave the Start page field in the manifest blank and let the activation happen through your default `activated` handler in the same script context as when the app is run normally.

In this case the `activationKind` value in the event will be `search`, a case that you want to handle separately from `launch`. To see this in action, let's turn to the [Search app contract sample](#), which I'll just refer to as the search sample since its given name is grammatically odd! Its activation code is found in `js/default.js`—code that's applicable to the entire app:

```
function activated(eventObject) {
    if (eventObject.detail.kind === Windows.ApplicationModel.Activation.ActivationKind.launch) {
        eventObject.setPromise(WinJS.UI.processAll().then(function () {
            var url = WinJS.Application.sessionState.lastUrl || scenarios[0].url;
            return WinJS.Navigation.navigate(url);
        }));
    } else if (eventObject.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.search) {
        eventObject.setPromise(WinJS.UI.processAll().then(function () {
```

```

        if (eventObject.detail.queryText === "") {
            // Navigate to your landing page since the user is pre-scoping to your app.
        } else {
            // Display results in UI for eventObject.detail.queryText and
            // eventObject.detail.language (that represents user's locale).
        }

        // Navigate to the first scenario since it handles search activation.
        var url = scenarios[0].url;
        return WinJS.Navigation.navigate(url, { searchDetails: eventObject.detail });
    });
}
}

```

In the search activation path, it's clearly good to avoid any processing that isn't needed by the search page itself, but you still need to be prepared to navigate to other parts of the app when a result is chosen. Also, if the app is being launched in response to a search, be sure to reload both general settings as you would with a normal launch as well as session state when `previousExecutionState` is `terminated`. This means, in fact, that the state of a results page is part of the app's session state; you'll normally want to save the last search term as part of that state so that you can rehydrate the results page when needed.

The sample doesn't actually search any real data—it just outputs messages when certain events happen. But you can test this activation path in a couple of ways. First, if the app isn't running, invoke the search charm, enter some query text, and then select the search sample. You'll find that it ends up on the page for Scenario 1 and shows the search term right away. This tells you that it processed the activation and picked up the search term from `eventObject.detail.queryText`, as you can see in the code above. (Look also at `js/scenario1.js` where it outputs the term in the page's `processed` method.)

To step through the same code, set a breakpoint within the `searchTarget` case of the `activated` handler and run the app in the Visual Studio debugger. Invoke the search charm, enter a query, select some other app (which will do a search), and then switch back to the sample. You should hit your breakpoint as the activated handler will be called with the activation kind of `search`.

When activated through search, be sure that the page gets fully processed with calls like `WinJS.UI.processAll`. (You don't need to worry if the app is already running; `processAll` won't do redundant work.)

It is important when your app is activated—as with handling `querysubmitted` and/or `querychanged` events—to note that the `queryText` might be empty. In this case you can show default results or navigate to your home page if that's more appropriate. See "Sidebar: Testing Search."

Sidebar: Testing Search

A number of variations with the Search charm can affect how a search target app is launched and with what parameters. To be sure that you've exercised all applicable code paths, be sure to test these conditions:

- App is running and search is invoked with no query text, query text with known results, and query text that returns no results.
- App is not running and is invoked from the search charm, with all the variations on text listed above.
- App is in the snapped state and is invoked as above, in which case the app should unsnap.
- App is suspended and is invoked as above.

You should also be mindful of how you present results, taking care that the primary results are not hidden by the Search pane, which will remain visible until the user dismisses it.

Sidebar: Synchronizing In-App Search with the Search Pane

Some types of apps will still maintain their own in-app search UI in addition to using the search pane, or in other ways they might have some kind of search term that would be good to keep in sync with the term shown in the search pane. To do this, the app can ask the search pane for its `queryText` value and can attempt to set that value through the [SearchPane.trySetQueryText](#) method. This call will fail, mind you, if the app isn't itself visible or if the search pane is already visible or becoming visible.

Providing Query Suggestions

Using `querysubmitted` and the activation sequence in the previous section gives you the basic level of search interaction, and Windows will automatically provide a history of the user's recent searches. Still, with just a little more work you can make the experience much richer. Because writing the code to actually perform the search, process the results, and display them beautifully is the bulk of the work with the Search contract anyway, adding support for query suggestions (this section) and result suggestions (in the next section) is a relatively small investment with a huge impact on the overall user experience.

To go beyond the default search history and provide as-the-user-is-typing query suggestions, which appear to the user as shown on the left side of Figure 12-10, you have two options. Which one you use depends on what you want to suggest and the data that you're searching.

First, to provide suggestions from folders on the file system, such as the music, pictures, and videos libraries, the search pane provides a built-in implementation through its [setLocalContentSuggestionsSettings](#) method with results like those in Figure 12-12. As shown in Scenario 4 of the sample, you first create a [Windows.ApplicationModel.Search.LocalContentSuggestionSettings](#) object, populate its properties, and then pass that object to [setLocalContentSuggestionsSettings](#) (js/scenario4.js):

```
var page = WinJS.UI.Pages.define("/html/scenario4.html", {
    ready: function (element, options) {
```

```

var localSuggestionSettings = new
    Windows.ApplicationModel.Search.LocalContentSuggestionSettings();
localSuggestionSettings.enabled = true;
localSuggestionSettings.locations.append(Windows.Storage.KnownFolders.musicLibrary);
localSuggestionSettings.aqsFilter = "kind:=music";

Windows.ApplicationModel.Search.SearchPane.getForCurrentView()
    .setLocalContentSuggestionSettings(localSuggestionSettings);
}
});

```

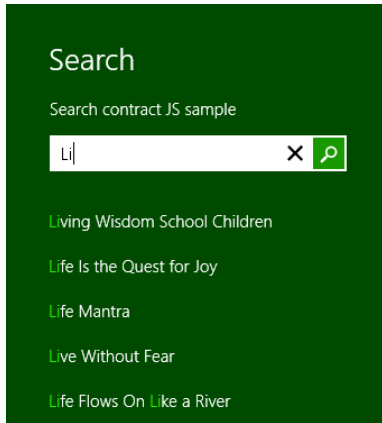


FIGURE 12-12 Suggestions from local folders as automatically provided by the search pane.

In populating the `LocalContentSuggestionSettings` properties, be sure first to set `enabled` to `true`. The `locations` collection (a vector) contains one or more `StorageFolder` objects to indicate where the search should take place. Because enumerating files to provide suggestions requires programmatic access to those folders, you need to make sure your app has the appropriate capabilities set in its manifest or that it has obtained programmatic access through the file picker. In the latter case, the app would provide UI elsewhere to configure the search locations (perhaps through the Settings pane, for instance).

You can also specify an [Advanced Query Syntax](#) (AQS) string in the `aqsFilter` property and/or some number of [Windows Properties](#) (like `System.Title`) within `propertiesToMatch` (a string vector). This is typically used to filter file types (as when searching a folder), but it can be as specific as you need it to be. For more on AQS, refer to “Rich Enumeration with File Queries” in Chapter 8, “State, Settings, Files, and Documents”; for more on Windows properties, refer to “Media File Metadata” in Chapter 10, “Media.”

As for the second option, `LocalContentSuggestionSettings` can do a lot for you, but clearly many apps will be searching on some other data source (whether local or online) and will thus need to supply suggestions from those sources. In these cases, listen for and handle the search pane’s `suggestionsrequested` event. Its `eventArgs` will contain the `queryText`, `language`, and `lunquisticDetails` as always, and in response you populate a collection of up to five suggestions in

the `eventArgs.request.searchSuggestionCollection`. Ideally this takes half a second or less, and it's important to know that all the results need to be in the collection once you return from your event handler.

Here's how it's done in Scenario 2 of the search app sample (where `suggestionList` is just a hard-coded list of city names):

```
Windows.ApplicationModel.Search.SearchPane.getForCurrentView()
.onsuggestionsrequested = function (eventObject) {
    var queryText = eventObject.queryText;
    var suggestionRequest = eventObject.request;
    var query = queryText.toLowerCase();
    var maxNumberOfSuggestions = 5;
    for (var i = 0, len = suggestionList.length; i < len; i++) {
        if (suggestionList[i].substr(0, query.length).toLowerCase() === query) {
            suggestionRequest.searchSuggestionCollection.appendQuerySuggestion(suggestionList[i]);
            if (suggestionRequest.searchSuggestionCollection.size === maxNumberOfSuggestions) {
                break;
            }
        }
    }
};
```

So if `query` contains "ba" as it would in Figure 12-10, the first 5 names in `suggestionList` will be Bangkok, Bangalore, Baghdad, Baltimore, and Bakersfield. Of course, a real app will be drawing suggestions from its own database or from a service (simulated in Scenarios 5 and 6, by the way), but you get the idea.⁵⁸ With a service, though, you should also check the `suggestionResult.isCanceled` property before starting a new request: this flag indicates that the search query hasn't actually changed from a previous query and it's not necessary to create new suggestions.

Note When the `SearchPane.searchHistoryEnabled` property is `true` (the default), a user's search history will be automatically tracked with prior searches appearing as suggestions when the search charm is first invoked (before the user types any other characters). Setting this property to `false` will disable the behavior, in which case an app can maintain its own history of previous `queryText` values. If an app does this, we recommend providing a means to clear the history through the app's Settings.

Apps can also use the `SearchPane.searchHistoryContext` property to create different histories depending on different contexts. When this value is set prior to the search charm being invoked, automatically managed search terms (`searchHistoryEnabled` is `true`) will be saved for that context. This has no effect when an app manages its own history, in which case it can manage different histories directly.

Now the `eventArgs.request` property, a `SearchPaneSuggestionsRequest` object has a few features you want to know about. Its `searchSuggestedCollection` property is unique—it's not an array or other generic vector but a `SearchSuggestionCollection` object with a `size` property and four methods: `appendQuerySuggestion` (to add a single item to the list, as shown above),

⁵⁸ Scenario 3 of the sample shows working with suggestions for East Asian languages using an IME.

[appendQuerySuggestions](#) (to add an array of items at once, as you might receive from a query to a service), [appendResultSuggestion](#) (see next section) and [appendSearchSeparator](#) (which is used to group suggestions). In the latter case, a separator is given a label and appears as follows:



The request object also has a [getDeferral](#) method if you need to perform an asynchronous operation to retrieve your suggestions. It works like all other deferral's we've seen: before starting the async operation (like [WinJS.xhr](#)), call [getDeferral](#) to retrieve the deferral object, start the operation, return from the [suggestionsrequested](#) method, and call the deferral's [complete](#) method inside the completed handler given to the async operation's [done](#) method. This is demonstrated again in Scenarios 5 and 6 of the sample since this would clearly be needed when querying a service for this purpose (derived from `js/scenario5.js`):

```
Windows.ApplicationModel.Search.SearchPane.getForCurrentView().onsuggestionsrequested =
function (eventObject) {
    var queryText = eventObject.queryText;
    var suggestionRequest = eventObject.request;

    var deferral = suggestionRequest.getDeferral();

    // Create request to obtain suggestions from service and supply them to the Search Pane.
    // Depending on the design of the service, you might vary the URL based on eventObject.language.
    // You might also compose queryText in the URL to let the service do the filtering.
    xhrRequest = WinJS.xhr({ url: /* URL to suggestion service */ });
    xhrRequest.done(
        function (request) {
            if (request.responseText /* or responseXML */) {
                // Populate suggestionRequest.searchSuggestionCollection based on response
            }

            deferral.complete(); // Indicate we're done supplying suggestions.
        },
        function (error) {
            // Call complete on the deferral when there is an error.
            deferral.complete();
        }
    );
};
```

You can use any JSON or XML response format you want, but since your app is doing the parsing, there are existing standards for returning search suggestions. For JSON, refer to the [OpenSearch Suggestions specification](#) and Scenario 5 in the sample where a JSON response can be directly parsed into an array and passed in one call to [appendQuerySuggestions](#). For XML, refer to the [XML Search Suggestions Format Specification](#) and Scenario 6. In the latter case, a function named [generateSuggestions](#) provides a generic parser routine for such a response, and although the sample doesn't demonstrate using separators, URIs, and images in those suggestions, the [generateSuggestions](#) function shows how to parse them and send them onto [appendQuery-](#)

`Suggestion[s]` as well as `appendResultSuggestion`, which we'll see next.

Providing Result Suggestions

As shown in Figure 12-10 (on the right side), a search target app can provide suggested *results* and not just suggested queries. This is also accomplished by handling the search pane's `suggestionsrequested` event as described in the previous section, only make sure you use `suggestionRequest.searchSuggestionCollection.appendResultSuggestion` to populate the results and not `appendQuerySuggestion[s]` (`appendSearchSeparator` can still be used). You also then need to handle the search pane's `resultSuggestionChosen` event to handle the user's selection as a result and not as a query.

In other words, handling the `querysubmitted` event means that you're taking the query text and populating a list of results in your own page. Because of this, you'll be handling click or tap events for those items directly, navigating to the appropriate details page. The `resultSuggestionChosen` event tells you that the same thing has happened in the system-owned search pane with results that are shown there from your suggestions. You thus process the `resultSuggestionChosen` event in the same way that you would handle an item invocation in your own page. The `eventArgs.tag` property in this case will contain the tag you provide for the suggested result in the `appendResultSuggestion` call.

This method actually takes five arguments in this order:

- `text` The first line text to show in the search pane (as in Figure 12-10).
- `detailText` The second line of text for a search result (as in Figure 12-10) that is also used for tooltips.
- `tag` The string you want to receive in the `resultSuggestionChosen` event.
- `image` An `IRandomAccessStreamReference` for the image to display. The base size of this image is 40x40 for 100% scale, 56x56 for 140%, and 72x72 for 180%.
- `imageAlternateText` The `alt` attribute for the image.

As noted in the previous section, the `generateSuggestions` function found in `js/scenario6.js` of the sample provide a generic parser that turns XML search suggestions into the appropriate `appendResultSuggestion` calls, including the use of `Windows.Storage.Streams.RandomAccessStreamReference.createFromUri` to convert an image URI to the appropriate stream reference. Typically, such URIs point to a remote source where ideally you'd be able to ask your service for different sized images based on the resolution scaling.

Local `ms-appx://` and `ms-appdata://` URIs are also allowable using the appropriate `.scale-1x0` naming convention. You should always, in fact, have a default image for suggested results in your package (using an `ms-appx://` URI to refer to it when necessary); the system will not provide one for you.

Type to Search

The final feature of Search is the ability to emulate the “type to search” behavior of the Windows Start screen, where the user doesn’t explicitly invoke the Search charm. If you haven’t done it before and you have a computer with a physical keyboard, press the Windows key to return to the Start screen, and start typing some app name *without* invoking the search charm first. Voila! The search charm appears automatically with results immediately filtered and displayed. This is essentially the same behavior that the Start button provided in previous versions of Windows, but it’s now much more visually engaging!

To enable this in your own app, simply set the [SearchPane.showOnKeyboardInput](#) property to `true`. You can enable or disable the behavior at any time through this property. Generally speaking, we recommend providing this behavior on your app’s main page(s) and on search results pages, but not on other subsidiary pages where there can be other input controls, nor on details pages showing content for a single item, nor on pages that support an in-page find capability. For details, see [Guidelines for Enabling Type to Search](#).

Launching Apps: File Type and URI Scheme Associations

Developers of Windows 8 apps have often asked whether it’s possible for one app to launch another. The answer is yes, with some restrictions (aren’t you surprised!). First, apps can be launched only through a file type or URI association, not directly by name or path. To be specific, the only way for a Windows 8 app to launch another app—including desktop applications—is through the [Windows.System.Launcher](#) API that provides you with two choices:

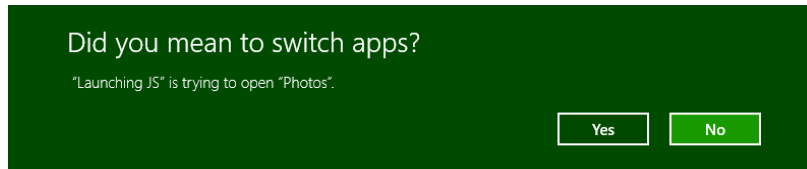
- `launchFileAsync` Launches another app associated with a given [StorageFile](#). An optional [LauncherOptions](#) object lets you specify a number of details (see below).
- `launchUriAsync` Launches another app associated with a given URI scheme, again with or without [LauncherOptions](#).

Note With both `launchFileAsync` and `launchUriAsync`, Windows 8 specifically blocks apps from launching any file or URI scheme that is handled by a system component and for which there is no legitimate scenario for a Windows 8 app to insert itself into that process. The [How to handle file activation](#) and [How to handle protocol activation](#) topics lists the specific file types and URI schemes in question.

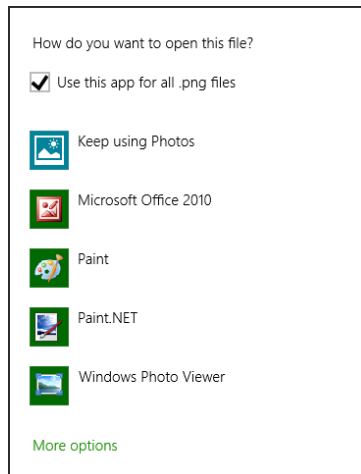
The result of both these async methods, as passed to your completed handler, is a Boolean: `true` if the launch succeeded, `false` if not. That is, barring a catastrophic failure such as a low memory condition where the async operation will outright fail, these operations normally report success to your completed handler with a Boolean indicating the outcome. You’ll get a `false` result, for example, if you try to launch a file with executable code or other files that are blocked for security reasons.

However, you cannot know ahead of time what the result will be. This is the reason for the [LauncherOptions](#) parameter, through which you can provide fallback mechanisms:

- The `treatAsUntrusted` option (a Boolean, default is `false`) will display a warning to the user that they'll be switching apps if they proceed (see image below). This is good to use when you're unsure about the source of the association, such as launching a URI found inside a PDF or other document, and want to prevent the user from experiencing a classic bait-and-switch!



- `displayApplicationPicker` (a Boolean, default is `false`) will let the user choose which app to launch as part of the process (see image below). Note that the UI allows the user to change the default app for subsequent invocations. Also, the `LauncherOptions.ui` property can be used to control the placement of the app picker.



- `preferredApplicationDisplayName` and `preferredApplicationPackageFamilyName` provide a suggestion to the user to acquire a specific app from the Windows Store if no other app is available to service the request.
- Similarly, `fallbackUri` specifies a URI to which the user will be taken if no app can be found to handle the request and you don't have a specific suggestion in the Windows Store.
- Finally, for `launchUriAsync`, the `contentType` option identifies the content type associated with a URI that controls which app is launched. This is primarily useful when the URI doesn't contain a specific scheme but simply refers to a file on a network using a scheme such as `http` or `file` that would normally launch a browser for file download. With `contentType`, the default app that's registered for that type, rather than the

scheme, will be launched. That app, of course, must be able to then use the URI to access the file. In other words, this option is a way to pass a URI, rather than a whole file, to a handler app that you know can work with that URI.

Scenarios 1 and 2 of the [Association Launching Sample](#) provide a demonstration of using these methods with some of the options so you can see their effects.

On the flip side, as demonstrated in Scenarios 3 and 4 of the same sample, is the question of how an app associates itself with a file type or URI scheme so that it can be launched in these ways. These associations constitute the File Activation contract and the Protocol Activation contract. In both cases the target app must declare the file types and/or URI schemes it wishes to service in its manifest and must then provide for those activation kinds, as we'll see in the following sections.

Again, file or protocol association is the *only* means through which a Windows 8 app can launch another, so there's no guarantee that you'll actually launch a specific app. Of course, the more unique and specific the file type or URI scheme, the less likely it is that a consumer would have multiple apps to handle the association or even that there would be many such apps in the Windows Store. Indeed, designing a unique URI scheme interface, where the scheme is fairly app-specific, is really the best way to have one Windows 8 app launch and delegate a task to another, since all kinds of data can be passed in the URI string itself. The Maps app in Windows 8, for example, supports a *bingmaps* scheme for accomplishing mapping tasks from other apps. You can imagine the same for a stocks app, a calendar app, an email app (beyond *mailto*), and so forth. If you create such a scheme and want other apps to use it, you'll certainly need to provide documentation for its usage details, which means that another app can implement the same scheme and thus offer itself as another choice in the Windows Store. So, there's no guarantee even with a very specific scheme that you can know for certain that you'll be launching another known app, but this is about as close as you can get to that capability.⁵⁹

File Activation

To declare file activation capability, first go to the Declarations section of the manifest and add a "File Type Associations" declaration, the Visual Studio UI for which is shown in Figure 12-13. Each file type can have multiple specific types (notice the Add New button under Supported File Types), such as a JPEG having .jpg and .jpeg file extensions. Note that some file types are disallowed for apps; see [How to handle file activation](#) for the complete list.

⁵⁹ In any case, it's a good idea to register your URI scheme with the Internet Assigned Numbers Authority ([IANA](#)). [RFC 4395](#) is the particular specification for defining new URI schemes.

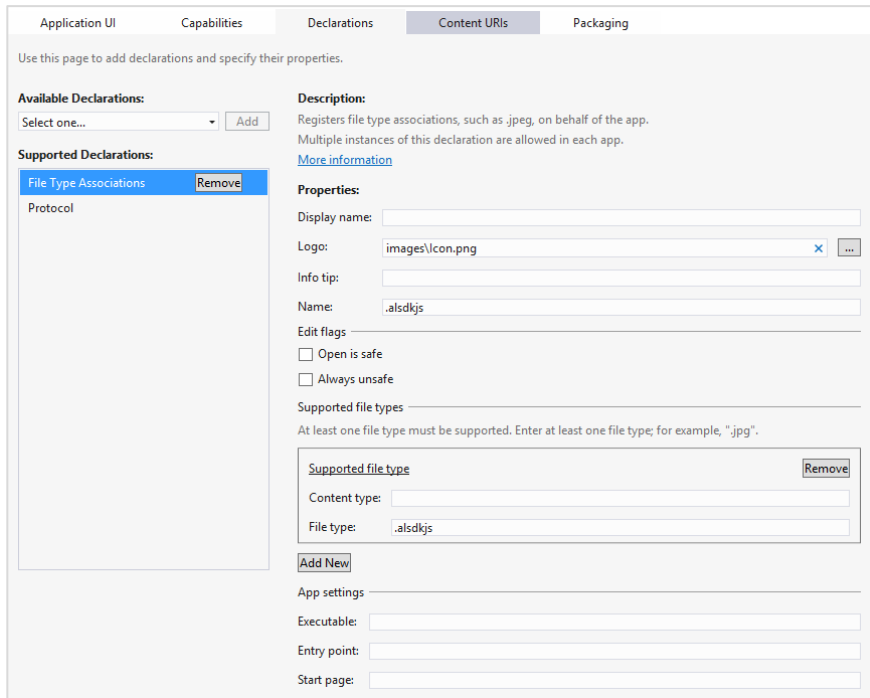


FIGURE 12-13 The Declarations > File Type Associations UI in the Visual Studio manifest designer.

Under Properties, the Display Name is the overall name for a group of file types (this is optional; not needed if you have only one type). The Name, on the other hand, is required—it’s the internal identity for the file group and one that should remain consistent for the entire lifetime of your app across all updates. In a way, the Name/Display Name properties for the whole group of file types is like your real name, and all the individual file types are nicknames—any of them ultimately refer to the core file type and your app.

Info Tip is tooltip text for when the user hovers over a file of this type and the app is the primary association. The Logo is a little tricky; in Visual Studio here, you simply refer to a base name for an image file, like you do with other images in the manifest. In your actual project, however, you should have multiple files for the same image in different target sizes (not resolution scales): 16x16, 32x32, 48x48, and 256x256. The [Association Launching Sample](#) uses such images.⁶⁰ These various sizes help Windows provide the best user experience across many different types of devices.

Under Edit Flags, these options control whether an “Open” verb is available for a downloaded file of this type: checking Open Is Safe will enable the verb; checking Always Unsafe disables the verb. Leaving both blank might enable the verb, depending on where the file is coming from and other settings

⁶⁰ Ignore, however, the sample’s use of targetsize-* naming conventions for the app’s tile images; target sizes apply only to file and URI scheme associations.

within the system.

At the very bottom of this UI you can also set a discrete start page for handling activations, but typically you'll use your main activation handler, as shown in `js/default.js` of the Association Launching sample (leading into `js/scenario3.js`).

There you'll receive the activation kind of `file`, in which case `eventArgs.detail` is a [WebUIFile-ActivatedEventArgs](#): its `files` property contains the array of `StorageFile` objects from `Windows.System.Launcher.launchFileAsync`, and its `verb` property will be `"open"`. You respond, of course, by opening and presenting the file contents in whatever way is appropriate to the app.

Of course, since the file the app is pointed to might have come from anywhere, treat it as untrusted content, as we mentioned earlier, for share targets. Avoid taking permanent actions based on those the file contents.

As with the Search contract, too, be sure to test file activation when the app is already running and when it must be started anew. In all cases be sure to load app settings and restore session state if `eventArgs.detail.previousExecutionState` is `terminated`.

Protocol Activation

Creating a URI scheme association for an app is much like a file type association. In the Declarations section of the manifest, add a Protocol declaration, as shown in Figure 12-14.

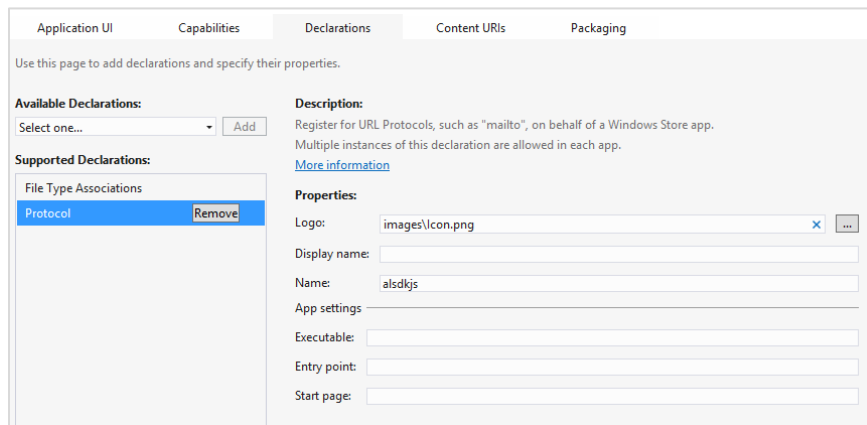


FIGURE 12-14 The Declarations > Protocol UI in the Visual Studio manifest designer.

Under Properties, the Logo, Display Name, and Name all have the same meaning as with file type associations (see the previous section). Similarly, while you can specify a discrete start page, you'll typically handle activation in your main activation handler, as demonstrated in again in `js/default.js` of the Association Launching sample (leading into `js/scenario4.js`).

There you'll receive the activation kind of `protocol`, in which case `eventArgs.detail` is a

[WebUIProtocolActivatedEventArgs](#): its `uri` property contains the URI from `Windows.System.Launcher.LaunchUriAsync`.

As I've mentioned with share targets and file activation, be warned that URIs with some unique scheme can come from anywhere, including potentially malicious sources. Be wary of any data in the URI, and avoid taking permanent actions with it. For instance, navigate to a new page, perhaps, but don't modify database records to try to `eval` anything in the URI.

Nevertheless, protocol associations are a primary way that an app can provide valuable services to others when appropriate. The built-in Maps app, for example, supports a `bingmaps://` URI scheme and association, so you can just launch a URI with the appropriate format to show the user a fully interactive map instead of trying to implement such capabilities yourself. This is similar to how you rely on an email client with the `mailto:` scheme; other kinds of apps can easily create a URI scheme interface for other services and workflows.

File Picker Providers

Back in Chapter 8, "State, Settings, Files, and Documents," we looked at how the file picker can be used to reference not only locations on the file system but also content that's managed by other apps or even created on-the-fly within other apps. Let's be clear on this point: the app that's *using* the file picker is doing so to obtain a `StorageFile` or `StorageFolder` for some purpose. But this does not mean that *provider* apps that can be invoked through the file picker necessarily manage their data *as* files. Their role is to take whatever kind of data they manage and package it up so that it *looks* like a file to the file picker.

In the "Using the File Picker" section of Chapter 8, for instance, we saw how the Windows 8 Camera app can be used to take a photo and return it through the file picker. Such a photo did not exist at the time the app was invoked; instead, it displayed its UI through which the user could essentially create a file that then gets passed back through the file picker. In this way, the Camera app shortcuts the whole process of creating a new picture, providing that function exactly when the user is trying to select a picture file. Otherwise the user would have to start the Camera app separately, take a photo, store it locally, and switch to the original app to invoke the file picker and relocate that new picture.

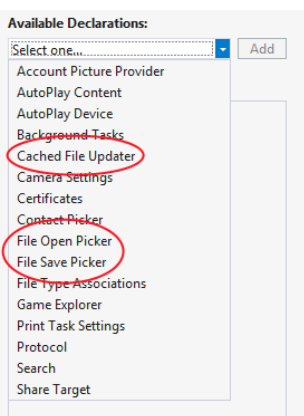
The file picker is not limited to pictures, of course: it works with any file type, depending on what the caller indicates it wants. One app might let the user go into a music library, purchase and download a track, and then return that file to the file picker. Another app might perform some kind of database query and return the results as a file, and still others might allow the user to browse online databases of file-type entities, again hiding the details of downloading and packaging that data as a file such that the user's experience of the file picker is seamless across the local file system, online resources, and apps that just create data dynamically. It's also possible to create an app that generates or acquires file-like data on the fly, such as the Camera app that allows you to take a picture or an audio app that could record a new sound. In such cases, however, note that the file picker contracts are designed for relatively quick in-and-out experiences. For this reason an app should provide only basic editing

capabilities (like cropping a photo or trimming the audio) in this context.

As with the Search and Share target contracts, Visual Studio and Blend provide an item template for file picker providers, specifically the File Open Picker contract item in the Add > New Item dialog as we've seen before (it's hiding off the top of the list in Figure 12-4). This gives you a basic selection structure built around a Listview control, but not much else. For our purposes here we won't be using this template and we'll draw on samples instead. Generally speaking, when servicing the file picker contracts, an app should use the same views and UI as it does when launched normally, thereby keeping the app experience consistent in both scenarios.

Manifest Declarations

To be a provider for the file picker, an app starts by—what else!—adding the appropriate declaration to its manifest. In this case there are actually three declarations: File Open Picker, File Save Picker, and Cached File Updater, as shown below in Visual Studio's manifest designer. Each of these declarations can be made once within any given app.



The File Open Picker and File Save Picker declarations are what make a provider app available in the dialogs invoked through the [Windows.Storage.Pickers.FileOpenPicker](#) and [FileSavePicker](#) API. The calling app in both cases is completely unaware that another app might be invoked—all the interaction is between the picker and the provider app through the contract, with the latter being responsible for first displaying a UI through which to select an object and second for returning a [StorageFile](#) object for that item.

With both the File Open Picker and File Save Picker contracts, the provider app indicates in its manifest those file types that it can service through the Add New button in the image below; the file picker will then make that app available as a choice only when the calling app indicates a matching file type. The Supports Any File Type option that you see here will make the app always appear in the list, but this is appropriate only for apps like SkyDrive that provide a general storage location. Apps that work only with specific file types should indicate those types.

Properties:

Supported file types

At least one file type must be supported. Either select "Supports any file type", or enter at least one specific file type; for example, ".jpg".

☐ Supports any file type

Add New

App settings

Executable:

Entry point:

Start page:

Here you can see that the provider app also indicates a Start Page for the open and save provider contracts separately—the operations are distinct and independent. In both cases, as we’ve seen for other contracts, these are the pages that the file picker will load when the user selects this particular provider app. As with Share targets, these pages are typically independent of the main app and will have their own script contexts and activation handlers, as we’ll see in the next section. (Again, the Executable and Entry Point options are not used for apps written in HTML and JavaScript; those exist for apps written in other languages.)

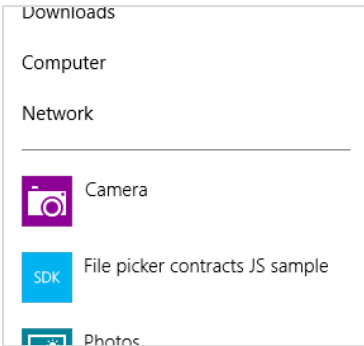
You might be asking: why are the open and save contracts separate? Won’t most apps generally provide both? Well, not necessarily. If you’re creating a provider app for a web service that is effectively read-only (like the image results from a search engine), you can serve only the file open case. If the service supports the creation of new files, such as a photo or document management service would, then you can also serve the file save case. There might also be scenarios where the provider would serve only the save case, such as writing to a sharing service. In short, Windows cannot presume the nature of the data sources that provider apps will work with, so the two contracts are kept separate.

While the next main section in this chapter covers the Cached File Updater contract, it’s good to know how it relates to the others here. This contract allows a provider app to synchronize local and remote copies of a file, essentially to subscribe to and manage change/access notifications for provided files. This is primarily of use to apps that represent a file repository (that is, one where the user will frequently open and save files, like SkyDrive or a database app). It’s essentially a two-way binding service for files when either local or remote copies can be updated independently. As such, it’s always implemented in conjunction with the file picker provider contracts.

Tip As noted earlier in this chapter, the [Sharing and exchanging data](#) topic on the Windows Developer Center has some helpful guidance as to when you might choose to be a provider for the file save picker contract and when being a share target is more appropriate.

Activation of a File Picker Provider

Demonstrations of the file picker provider contracts—for open and save—are found in the [Provide files and a save location sample](#), which I’ll refer to as the *provider sample*. Declarations for both are included in the manifest with Supports Any File Type, so the sample will be listed with other apps in all file pickers, as shown here:



When invoked, the Start page listed in the manifest for the appropriate contract (open or save) is loaded. These are fileOpenPicker.html and fileSavePicker.html, found in the root of the project. Both of these pages are again loaded independently of the main app and appear as shown in Figures 12-15 and 12-16. Note that the title of the app and the color scheme is determined by the Application UI settings in the provider app’s manifest. In particular, the text comes from the Display Name field and the colors come from the Foreground Text and Background Color settings under Tile, as shown in Figure 12-17. Note that the system automatically adds the down chevron (v) next to the title in Figures 12-15 and 12-16 through which the user can select a different picker location or provider app.

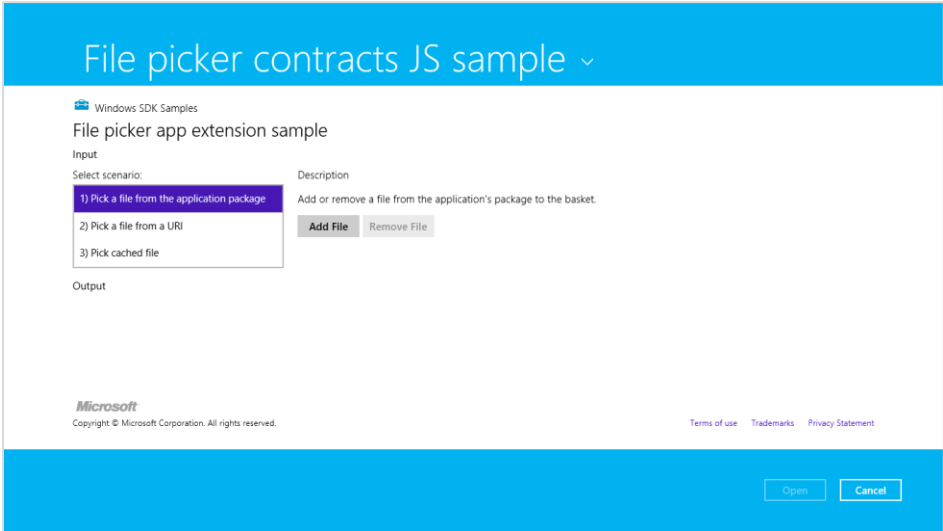


FIGURE 12-15 The Open UI as displayed by the sample.

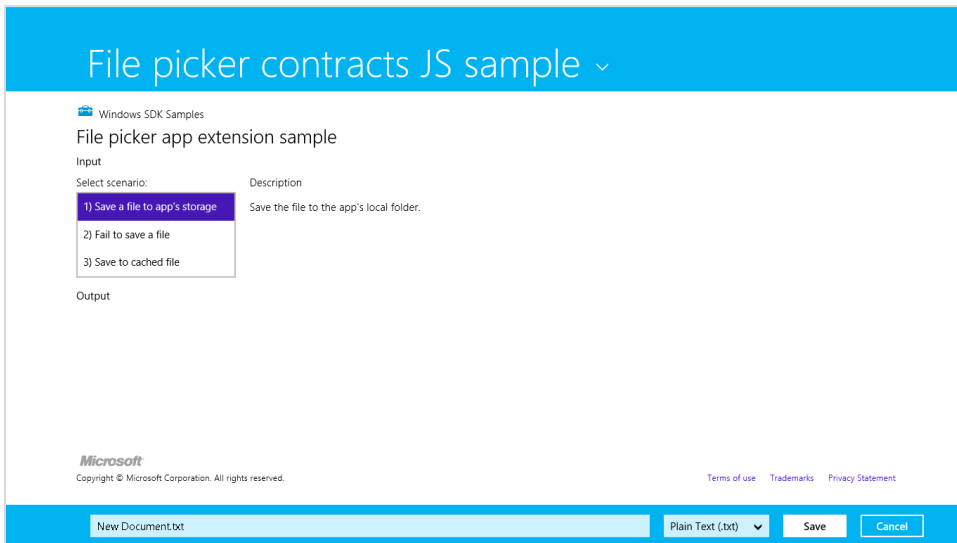


FIGURE 12-16 The Save UI as displayed by the sample.

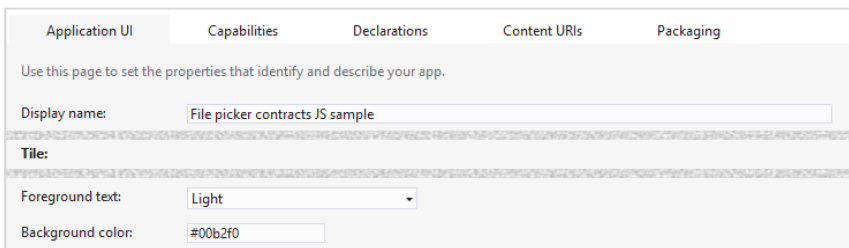


FIGURE 12-17 Application UI settings in the manifest that affect the appearance of the open and save picker UI for a provider app. The gray bars in this image represent other fields that I've omitted for brevity.

When you first run this sample, you won't see either of these pages. Instead you'll see a page through which you can invoke the file open or save pickers and then choose this app as a provider. You can do this if you like, but I recommend using a different app to invoke the pickers, just so we're clear on which app is playing which role. For this purpose you can use the sample we used in Chapter 8, the [Access and save files using the file picker sample](#). You can even use something like the Windows 8 Music app where the Open File command on its app bar will invoke a picker wherein this provider sample will be listed.

Whatever your choice, the important parts of the provider sample are its separate pages for servicing its contracts, which are again `fileOpenPicker.html` and `fileSavePicker.html`. In the first case, the code is contained in `js/fileOpenPicker.js` where we can see the `activated` event handler with the activation kind of `fileOpenPicker`:

```
function activated(eventObject) {
    if (eventObject.detail.kind ===
```

```

Windows.ApplicationModel.Activation.ActivationKind.fileOpenPicker) {
    fileOpenPickerUI = eventObject.detail.fileOpenPickerUI;

    eventObject.setPromise(WinJS.UI.processAll().then(function () {
        // Navigate to a scenario page...
    }));
}
}

```

Here `eventObject.detail` is a [WebUIFileOpenPickerActivatedEventArgs](#) object, whose `fileOpenPickerUI` property (a [Windows.Storage.Pickers.Providers.FileOpenPickerUI](#) object) provides the means to fulfill the provider's responsibilities with the contract.

In the second case, the code is in `js/fileSavePicker.js` where the activation kind is `fileSavePicker`:

```

function activated(eventObject) {
    if (eventObject.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.fileSavePicker) {
        fileSavePickerUI = eventObject.detail.fileSavePickerUI;

        eventObject.setPromise(WinJS.UI.processAll().then(function () {
            // Navigate to a scenario page
        }));
    }
}

```

where `eventObject.detail` is a [WebUIFileSavePickerActivatedEventArgs](#) object. As with the open contract, the `fileSavePickerUI` property of this (a [Windows.Storage.Pickers.Providers.-FileSavePickerUI](#) object) provides the means to fulfill the provider's side of the contract.

In both open and save cases, the contents of the contract's Start page is displayed within the letterboxed area between the system-provided top and bottom bands. If that content overflows the provided space, scrollbars would be provided only within that area—the top and bottom bands always remain in place. In both cases, also provide the usual features for activation, such as the `splashScreen` and `previousExecutionState` properties, just as we saw in Chapter 3, "App Anatomy and Page Navigation," meaning that you should reload necessary session state and use extended splash screens as needed.

What's most interesting, though, are the contract-specific interactions that are represented in the different scenarios for these pages (as you can see in Figures 12-15 and 12-16). Let's look at each in turn.

Note For specific details on designing a file picker experience, see [Guidelines for file pickers](#).

File Open Provider: Local File

The provider for file open works through the `FileOpenPickerUI` object supplied with the `fileOpenPicker` activation kind. Simply said, whatever kind of UI the provider offers to select some file or data will be wired to the various methods, properties, and events of this object.

First, the UI will use the `allowedFileTypes` property to filter what it displays for selection—clearly, the provider should not display items that don't match what the file picker is being asked to pick! Next, the UI can use the `selectionMode` property (a `FileSelectionMode` value) to determine if the file picker was invoked for `single` or `multiple` selection.

When the user selects an item within the UI, the provider calls the `addFile` method with the `StorageFile` object as appropriate for that item. Clearly, the provider has to somehow create that `StorageFile` object. In the sample's open picker > Scenario 1, this is accomplished with a `StorageFolder.getFileAsync` (where the `StorageFolder` is the package location).

```
Windows.ApplicationModel.Package.current.installedLocation
    .getFileAsync("images\\squareTile-sdk.png").then(function (fileToAdd) {
    addFileToBasket(localFileId, fileToAdd);
})
```

where `addFileToBasket` just calls `FileOpenPickerUI.addFile` and displays messages for the result. That result is a value from `Windows.Storage.Pickers.Provider.AddFileResult`: `added` (success), `alreadyAdded` (redundant operations, so the file is already there), `notAllowed` (adding is denied due to a mismatched file type), and `unavailable` (app is not visible). These really just help you report the result to users in your UI. Note also that the `canAddFile` method might be helpful for enabling or disabling add commands in your UI as well, which will prevent some of these error cases from ever arising in the first place.

The provider app must also respond to requests to remove a previously added item, as when the user removes a selection from the “basket” in the multi-select file picker UI. To do this, listen for the `FileOpenPickerUI` object's `fileRemoved` event, which provides a file ID as an argument. You pass this ID to `containsFile` followed by `removeFile` as in the sample (`js/fileOpenPickerScenario1.js`):

```
// Wire up the event in the page's initialization code
fileOpenPickerUI.addEventListener("fileremoved", onFileRemovedFromBasket, false);

function removeFileFromBasket(fileId) {
    if (fileOpenPickerUI.containsFile(fileId)) {
        fileOpenPickerUI.removeFile(fileId);
    }
}
```

If you need to know when the file picker UI is closing your page (such as the user pressing the Open or Cancel buttons as shown in Figure 12-15), listen for the `closing` event. This gives you a chance to close any sessions you might have opened with an online service and otherwise perform any necessary cleanup tasks. In the `eventArgs` you'll find an `isCanceled` property that indicates whether the file picker is being canceled (`true`) or if it's being closed due to the Open button (`false`). The `eventArgs.closingOperation` object also contains a `getDeferral` method and a `deadline` property that allows you to carry out async operations as well, similar to what we saw in Chapter 3 for the `suspending` event.

A final note is that a file picker provider should respect the `FileOpenPickerUI.settingsIdentifier` to relaunch the provider to a previous state (that is, a previous picker session). If you remember from

the other side of this story, an app that's using the file picker can use the `settingsIdentifier` to distinguish different use cases within itself—perhaps to differentiate certain file types or feature contexts. The identifier can also differ between different apps that invoke the file picker. By honoring this property, then, a provider app can maintain a case-specific context each time it's invoked (basically using `settingsIdentifier` in filenames and the names of settings containers), which is how the built-in file pickers for the file system works.

It's also possible for the provider app to be suspended while displaying its UI and could possibly be shut down if the calling app is closed. However, if you manage picker state based on `settingsIdentifier` values, you don't need to save or manage any other session state where your picker functionality is concerned.

File Open Provider: URI

For the most part, Scenario 2 of the open file picker case in the provider sample is just like we've seen in the previous section. The only difference is that it shows how to create a `StorageFile` from a nonfile source, such as an image that's obtained from a remote URI. In this situation we need to obtain a data stream for the remote URI and convert that stream into a `StorageFile`. Fortunately, a few WinRT APIs make this very simple, as shown in `js/fileOpenPickerScenario2.js` within its `onAddFileUri` method:

```
function onAddUriFile() {
    // Respond to the "Add" button being clicked
    var imageSrcInput = document.getElementById("imageSrcInput");

    if (imageSrcInput.value !== "") {
        var uri = new Windows.Foundation.Uri(imageSrcInput.value);
        var thumbnail = Windows.Storage.Streams.RandomAccessStreamReference.createFromUri(uri);

        // Retrieve a file from a URI to be added to the picker basket
        Windows.Storage.StorageFile.createStreamedFileFromUriAsync("URI.png", uri,
            thumbnail).then(function (fileToAdd) {
                addFileToBasket(uriFileId, fileToAdd);
            },
            function (error) {
                // ...
            });
    } else {
        // ...
    }
}
```

Here `Windows.Storage.StorageFile.createStreamedFileFromUriAsync` does the honors to give us a `StorageFile` for a URI, and `addFileToBasket` is again an internal method that just calls the `addFile` method of the `FileOpenPickerUI` object.

File Save Provider: Save a File

Similar to how the file open provider interacts with a `FileOpenPickerUI` object, a provider app for saving files works with the specific methods, properties, and events `FileSavePickerUI` class. Again, the

open and save contracts are separate concerns because the data source for which you might create a provider app might or might not support save operations independently of open. If you do support both, though, you will likely reuse the same UI and would thus use the same Start page and activation path.

Within the `FileSavePickerUI` class, we first have the `allowedFileTypes` as provided by the app that invoked the file save picker UI in the first place. As with open, you'll use this to filter what you show in your own UI so that users can clearly see what items for these types already exist. You'll also typically want to populate a file type drop-down list with these types as well.

For restoring the provider's save UI for the specific calling app from a previous session, there is again the `settingsIdentifier` property.

Referring back to Figure 12-16, notice the controls along the bottom of the screen, the ones that are automatically provided by the file picker when the provider app is invoked. When the user changes the filename field, the provider app can listen for and handle the `FileSavePickerUI` object's `filenameChanged` event; in your handler you can get the new value from the `fileName` property. If the provider app has UI for setting the filename, it cannot write to this property, however. It must instead call `trySetFileName`, whose return value from the `SetFileNameResult` enumeration is either `succeeded`, `notAllowed` (typically a mismatched file type), or `unavailable`. This is typically used when the user taps an item in your list, where the expected behavior is to set the filename to the name of that item.

The most important event, of course, happens when the user finally taps the Save button. This will fire the `FileSavePickerUI` object's `targetFileRequested` event. You must provide a handler for this event, in which you must create an empty `StorageFile` object in which the app that invoked the file picker UI can save its data. The name of this `StorageFile` must match the `fileName` property.

The `eventArgs` for this event is a `Windows.Storage.Pickers.Providers.TargetFileRequested-EventArgs` object. This contains a single property named `request`, which is a `TargetFileRequest`. Its `targetFile` property is where you place the `StorageFile` you create (or `null` if there's an error). You must set this property before returning from the event handler, but of course you might need to perform asynchronous operations to do this at all. For this purpose, as we've seen many times, the request also contains a `getDeferral` method. This is used in Scenario 1 of the provider sample's save case (`js/fileSavePickerScenario1.js`):

```
function onTargetFileRequested(e) {
    var deferral = e.request.getDeferral();

    // Create a file to provide back to the Picker
    Windows.Storage.ApplicationData.current.localFolder.createFileAsync(fileSavePickerUI.fileName)
        .done(function (file) {
            // Assign the resulting file to the targetFile property and complete the deferral
            e.request.targetFile = file;
            deferral.complete();
        }, function () {
            // Set the targetFile property to null and complete the deferral to indicate failure
```

```

        e.request.targetFile = null;
        deferral.complete();
    });
};

```

In your own app you will, of course, replace the `createFileAsync` call in the local folder with whatever steps are necessary to create a file or data object. Where remote files are concerned, on the other hand, you'll need to employ the Cached File Updater contract (see “Cached File Updater” below).

File Save Provider: Failure Case

Scenario 2 of the provider sample's save UI just shows one other aspect of the process: displaying errors in case there is a real failure to create the necessary `StorageFile`. Generally speaking, you can use whatever UI you feel is best and consistent with the app in general, to let the user know what they need to do—typically with a `MessageDialog` as does the sample:

```

function onTargetFileRequestedFail(e) {
    var deferral = e.request.getDeferral();

    var messageDialog = new Windows.UI.Popups.MessageDialog("If the app needs the user to correct
a problem before the app can save the file, the app can use a message like this to tell
the user about the problem and how to correct it.");

    messageDialog.showAsync().done(function () {
        // Set the targetFile property to null and complete the deferral to indicate failure once
        // the user has closed the dialog. This will allow the user to take any necessary
        // corrective action and click the Save button once again.
        e.request.targetFile = null;
        deferral.complete();
    });
};

```

Cached File Updater

Using the cached file updater contract provides for keeping local copies of a file in sync with one managed by a provider app, most often a remote file of some kind. This contract is specifically meant for apps (such as the SkyDrive app in Windows) that serve primarily as a storage location for user data—a place where users regularly save, access, and update files. In other cases where the user is generally going to pick a file and use it some scenario but not otherwise come back to it, using the file picker contracts is entirely sufficient.

Back in Chapter 8, we saw some of the method calls that are made by an app that uses the file picker: `Windows.Storage.CachedFileManager.deferUpdates` and `Windows.Storage.CachedFileManager.completeUpdatesAsync`. This usage is shown in Scenarios 4 and 6 of the [Access and save files using the file picker sample](#) we worked with in that chapter. Simply said, these are the calls that a file-consuming app makes if and when it makes updates to a file that it obtained from a file picker, since it won't know (and shouldn't care) whether the file provider has

another copy in database, web service, etc., that needs to be kept in sync. If the provider needs to handle synchronization, the consuming app's calls to these methods will trigger the necessary cached file updater UI of the provider app, which might or might not be shown, depending on the need (hopefully very seldom!). Even if the consuming app doesn't call these methods, the provider app will still be notified of changes but won't be able to show any UI.

There are two directions or *targets* with which this contract works, depending on whether it's needed to update a *local* (cached) copy of a file or the *remote* (source) copy. In the first case, the provider is asked to update the local copy, typically when the consuming app attempts to access that file (pulling it from the [FutureAccessList](#) or [MostRecentlyUsed](#) list of [Windows.Storage.AccessCache](#)). In the second case, the consuming app has modified the file such that the provider needs to propagate those changes to its source copy.

From a provider app's point of view, the need for such updates comes into play whenever it supplies a file to another app. This can happen through the file picker contracts, as we've seen in the previous section, but also through file type associations as well as the share contract. In the latter case a share source app is, in a sense, a file provider and might make use of the cached file updater contract as well. In short, if you want your file-providing app to be able to track and synchronize updates between local and remote copies of a file, this is the contract to use.

Supporting the contract begins with a manifest declaration, of course, as shown below, where the Start page indicates the page implementing the cached file updater UI. (Again, apps written in HTML/JavaScript do not use the Executable and Entry Point fields.) That page will handle the necessary events to update files and might or might not actually be displayed to the user, as we'll see later.

The screenshot shows the Windows 8 manifest editor interface. On the left, under 'Available Declarations', there is a dropdown menu with 'Select one...' and an 'Add' button. Below this, under 'Supported Declarations', a list contains 'Cached File Updater', 'File Open Picker', and 'File Save Picker'. The 'Cached File Updater' entry has a 'Remove' button next to it. On the right, the 'Description' section states: 'Registers the app as a cached file updater, allowing the app to provide updates to files that are accessed by other Windows 8 apps. Only one instance of this declaration is allowed per app.' Below the description is a 'More information' link. The 'Properties' section includes 'App settings', 'Executable' (empty text box), 'Entry point' (empty text box), and 'Start page' (containing 'cachedFileUpdater.html').

The next step for the provider is to indicate when a given [StorageFile](#) should be hooked up with this contract. It does so by calling [Windows.Storage.Provider.CachedFileUpdater.setUpdateInformation](#) on a provided file as shown in Scenario 3 of the [Provide files and a save location sample](#), which I'll again refer to as the *provider sample* for simplicity (js/fileOpenPickerScenario3.js):

```
function onAddFile() {
    // Respond to the "Add" button being clicked

    Windows.Storage.ApplicationData.current.localFolder.createFileAsync("CachedFile.txt",
        Windows.Storage.CreationCollisionOption.replaceExisting).then(function (file) {
        Windows.Storage.FileIO.writeTextAsync(file, "Cached file created...").then(function () {
```

```

        Windows.Storage.Provider.CachedFileUpdater.setUpdateInformation(file, "CachedFile",
            Windows.Storage.Provider.ReadActivationMode.beforeAccess,
            Windows.Storage.Provider.WriteActivationMode.notNeeded,
            Windows.Storage.Provider.CachedFileOptions.requireUpdateOnAccess);
        addFileToBasket(localFileId, file);
    }, onError);
}, onError);
};

```

Note `setUpdateInformation` is within the `Windows.Storage.Provider` namespace and is different from the `Windows.Storage.CachedFileManager` object that's used on the other side of the contract; be careful to not confuse the two.

The `setUpdateInformation` method takes the following arguments:

- A `StorageFile` for the file in question.
- A content identifier string that identifies the remote resource to keep in sync.
- A `ReadActivationMode` indicating whether the calling app can read its local file without updating it; values are `notNeeded` and `beforeAccess`.
- A `WriteActivationMode` indicating whether the calling app can write to the local file and whether writing triggers an update; values are `notNeeded`, `readOnly`, and `afterWrite`.
- One or more values from `CachedFileOptions` (that can be combined with bitwise-OR) that describes the ways in which the local file can be accessed without triggering an update; values are `none` (no update), `requireUpdateAccess` (update on accessing the local file), `useCachedFileWhenOffline` (will update on access if the calling app desires, and access is allowed if there's no network connection), and `denyAccessWhenOnline` (triggers an update on access and requires a network connection).

It's through this call, in other words, that the provider specifically controls how and when it should be activated to handle updates when a local file is accessed.

So, together we have two cases where the provider app will be invoked and might be asked to show its UI: one where the calling app updates the file, and another when the calling app accesses the file and might need an update before reading its contents.

Before going into the technical details, let's see how these interactions appear to the user. To see the cached file updater in action using the sample, we need to invoke it by using the file picker from another app. First, then, run the provider sample to make sure its contracts are registered. Then run the aforementioned [Access and save files using the file picker sample](#). In the latter, Scenarios 4, 5, and 6 cause interactions with the cached file updater contract. Scenarios 4 and 6 write to a file to trigger an update to the remote copy; Scenario 5 accesses a local file that will trigger a local update as part of the process.

Updating a Local File: UI

In Scenario 5 (updating a local file), start by tapping the Pick Cached File button in the UI shown here:

File picker JS sample

Input

Select scenario:

- 2) Pick multiple files
- 3) Pick a folder
- 4) Save a file
- 5) Open a cached file**
- 6) Update a cached file

Description

Demonstrates how an app that integrates with the Cached File Updater contract can provide the latest version of a file to your app. This scenario requires that you have the File Picker Contracts Sample app.

Steps:

- 1) Launch the file picker with the "Pick cached file" button
- 2) Select "File picker contracts sample" app from the list of locations
- 3) Select scenario "3) Pick cached file"
- 4) Click "Add file to basket" and click "Open" in the file picker
- 5) Click "Output latest version" button

Pick cached file Output latest version

This will launch the provider sample; in that view, select Scenario 3 so that you see the UI shown in Figure 12-18. This is the mode of the provider sample that is just a file picker *provider*, (`js/fileOpenPickerScenario3.js`) where it calls `setUpdateInformation`. This is *not* the UI for the cached file updater yet. Click the Add File to Basket button, and tap the Open button. This will return you to the first app (the picker sample in the above graphic) where the Output Latest Version button will now be enabled. Tapping *that* button will then invoke the provider sample through the cached file updater contract, as shown in Figure 12-19. This is what appears when there's a need to update the local copy of the cached file.

File picker contracts JS sample

Windows SDK Samples

File picker app extension sample

Input

Select scenario:

- 1) Pick a file from the application package
- 2) Pick a file from a URI
- 3) Pick cached file**

Description

Add or remove a cached file to the basket.

Add file to basket Remove file from basket

Output

File added to the basket.

Microsoft

Copyright © Microsoft Corporation. All rights reserved.

Terms of use Trademarks Privacy Statement

Open Cancel

FIGURE 12-18 The provider sample's UI for picking a file; the `setUpdateInformation` method is called on the provided file to set up the cached file updater relationship.

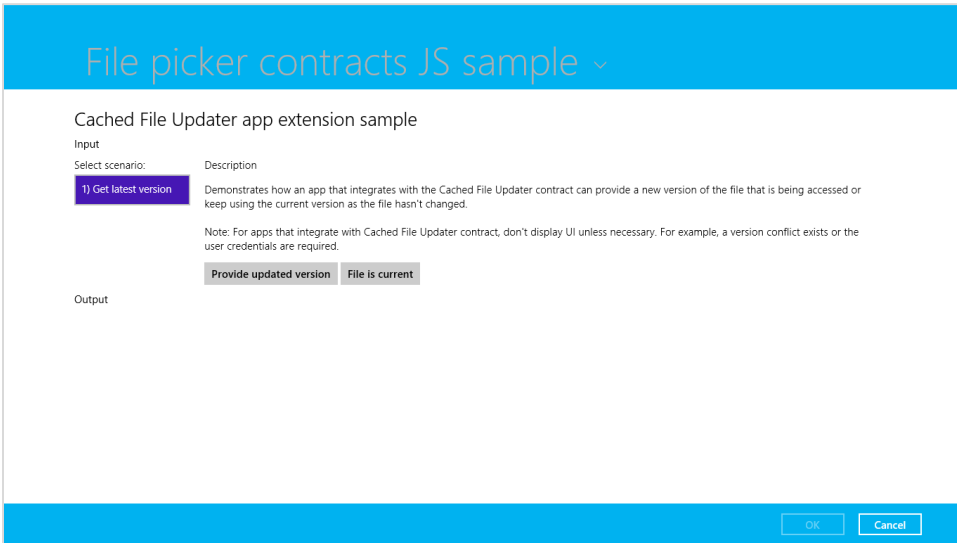
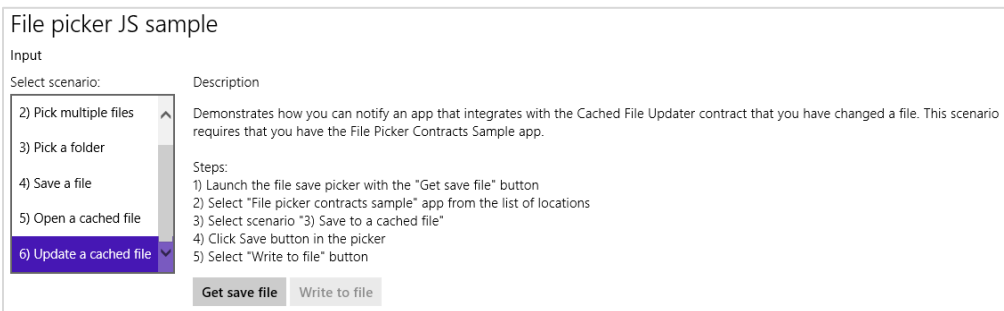


FIGURE 12-19 The cached file updater provider sample's UI for the cached file updater contract on a local file.

Take careful note of the description in the sample. While the sample shows this UI by default, a cached file updater app will not show it unless it's necessary to resolve conflicts or collect credentials. Oftentimes no such interaction is necessary and the provider silently provides an update to the local file or indicated that the file is current. The sample's UI here is simply providing both those options as explicit choices (and be sure to choose one of them because selecting Cancel will throw an exception).

Updating a Remote File: UI

In Scenario 6 (updating a remote file) of the file picker sample, we can see the interactions that take place when the consuming app writes changes to its local copy, thereby triggering an update to the remote copy. Start by tapping the Get Save File button in the UI shown next:



In the picker, select the provider sample as the picker source, which invokes the UI of Figure 12-20 through the file save picker contract and implemented through `html/fileSavePickerScenario3.html` and `js/fileSavePickerScenario3.js`. If you look in the JavaScript file, you'll again see a call to

`setUpdateInformation` that's called when you enter a file name and tap Save. Doing so also returns you to the picker sample above where Write to File should now be enabled. Tapping Write to File then reinvokes the provider sample through the cached file updater contract with the UI shown in Figure 12-21. This UI is intended to demonstrate how such a provider app would accommodate overwriting or renaming the remote file.

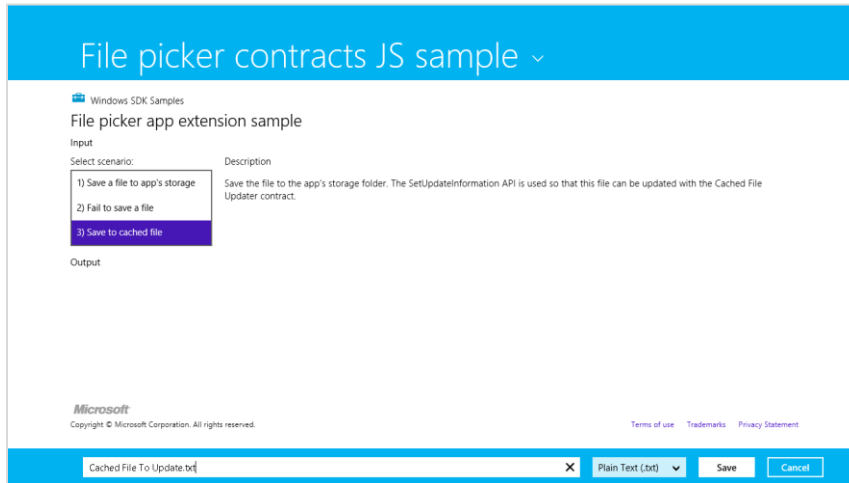


FIGURE 12-20 The provider sample's UI for saving a file; the `setUpdateInformation` method is again called on the provided file to set up the cached file updater relationship.

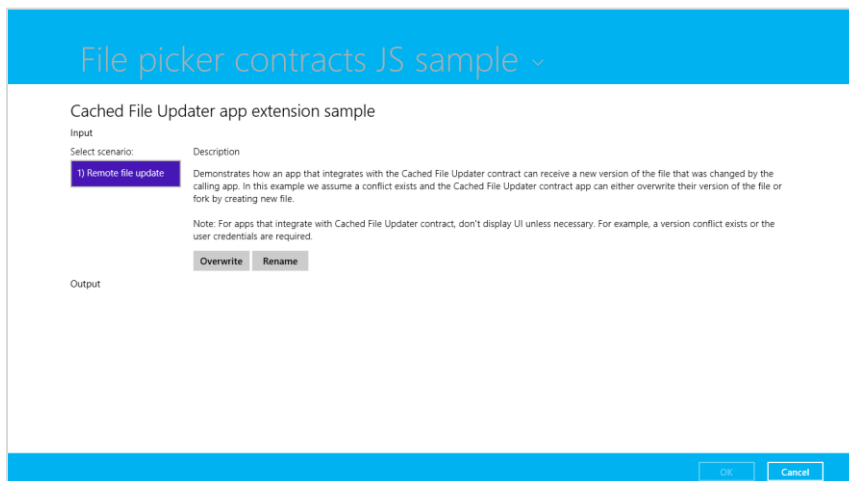


FIGURE 12-21 The cached file updater provider sample's UI for the cached file updater contract on a remote file.

Update Events

Let's see how the cached file updater contract looks in code now. As you will by now expect, the

provider app is launched and the Start page (`cachedFileUpdater.html` in the project root) loaded with the activation kind of `cachedFileUpdater`. This will happen for both local and remote cases, and as we'll see here, you use the same activation code for both. Here `eventObject.detail` is a [WebUICachedFileUpdaterActivatedEventArgs](#) that contains a `cachedFileUpdaterUI` property (a [CachedFileUpdaterUI](#)) along with the usual roster of `kind`, `previousExecutionState`, and `splashScreen`. Here's how it looks in `js/cachedFileUpdater.js` of the provider sample:

```
function activated(eventObject) {
    if (eventObject.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.cachedFileUpdater) {
        cachedFileUpdaterUI = eventObject.detail.cachedFileUpdaterUI;

        cachedFileUpdaterUI.addEventListener("fileupdaterequested", onFileUpdateRequest);
        cachedFileUpdaterUI.addEventListener("uirequested", onUIRequested);

        switch (cachedFileUpdaterUI.updateTarget) {
            case Windows.Storage.Provider.CachedFileTarget.local:
                // Code omitted: configures the sample to show cachedFileUpdaterScenario1 if needed.
                break;

            case Windows.Storage.Provider.CachedFileTarget.remote:
                // Code omitted: configures the sample to show cachedFileUpdaterScenario2 if needed.
                break;
        }
    }
}
```

When the provider app is invoked to update a local file from the remote source, the `cachedFileUpdaterUI.updateTarget` property will be `local`, as you can see above. When the app is being asked to update a remote file with local changes, the target is `remote`. All the sample does in these cases is point to either `html/cachedFileUpdaterScenario1.html` (Figure 12-19) or `html/cachedFileUpdaterScenario2.html` (Figure 12-21) as the update UI. We'll see how this works in a moment.

The UI is not actually shown initially. What happens first is that the `CachedFileUpdaterUI` object fires its [fileUpdateRequested](#) event to attempt a silent update. Here the `eventArgs` is a [FileUpdateRequestedEventArgs](#) object with a single `request` property ([FileUpdateRequest](#)), an object that you'll want to save in a variable that's accessible from your update UI.

If it's possible to silently update a local file, follow these steps:

- Because you'll likely be doing async operations to perform the update, obtain a deferral from `request.getDeferral`.
- To update the contents of the local file, use one of these options:
- If you already have a `StorageFile` with the new contents, just call [request.updateLocalFile](#). This is a synchronous call, in which case you do not need to obtain a deferral.

- The local file's `StorageFile` object will be in `request.file`. You can open this file and write whatever contents you need within it. This will typically start an async operation, after which you return from the event handler.
- To update the contents of a remote file, copy the contents from `request.file` to the remote source.
- Depending on the outcome of the update, set `request.status` to a value from `FileUpdateStatus`: `complete` (the copies are sync'd), `incomplete` (sync didn't work but the local copy is still available), `userInputNeeded` (the update failed for need of credentials or conflict resolution), `currentlyUnavailable` (source can't be reached, and the local file is inaccessible), `failed` (sync cannot happen now or ever, as when the source file has been deleted), and `completeAndRenamed` (the source version has been renamed, generally to resolve conflicts).
- If you asked for a deferral and processed the outcome within completed and error handlers, call the deferral's `complete` method to finalize the update.

Now the provider might know ahead of time that it can't do a silent update at all—a user might not be logged into the back-end service (or credentials are needed each time), there might be a conflict to resolve, and so forth. In these cases the event handler here should check the value of `cachedFileUpdaterUI.uiStatus` (a `UIStatus`) and set the `request.status` property accordingly:

- If the UI status is `visible`, switch to that UI and return from the event handler. Complete the deferral when the user has responded through the UI.
- If the UI status is `hidden`, set `request.status` to `userInputNeeded` and return. This will trigger the `CachedFileUpdaterUI.onuiRequested` event followed by another `fileUpdateRequested` event where `uiStatus` will be `visible`, in which case you'll switch to your UI.
- If the UI status is `unavailable`, set `request.status` to `currentlyUnavailable`.

You can see some this in the sample's `onFileUpdateRequest` handler; it really handles only the `uiStatus` check because it doesn't attempt silent updates at all (as described in the comments below):

```
function onFileUpdateRequest(e) {
    fileUpdateRequest = e.request;
    fileUpdateRequestDeferral = fileUpdateRequest.getDeferral();

    // Attempt a silent update using fileUpdateRequest.file silently, or call
    // fileUpdateRequest.updateLocalFile in the local case, setting fileUpdateRequest.status
    // accordingly, then calling fileUpdateRequestDeferral.complete(). Otherwise, if you know
    // that user action will be required, execute the following code.

    switch (cachedFileUpdaterUI.uiStatus) {
        case Windows.Storage.Provider.UIStatus.hidden:
            fileUpdateRequest.status = Windows.Storage.Provider.FileUpdateStatus.userInputNeeded;
            fileUpdateRequestDeferral.complete();
```

```

        break;
    case Windows.Storage.Provider.UIStatus.visible:
        // Switch to the update UI (configured in the activated event)
        var url = scenarios[0].url;
        WinJS.Navigation.navigate(url, cachedFileUpdaterUI);
        break;
    case Windows.Storage.Provider.UIStatus.unavailable:
        fileUpdateRequest.status = Windows.Storage.Provider.FileUpdateStatus.failed;
        fileUpdateRequestDeferral.complete();
        break;
    }
}

```

Again, if a silent update succeeds, the provider app's UI never appears to the user. In the case of the provider sample, since it never attempts to do a silent update, it always does the check on `uiStatus`. When the app was just launched to service the contract, we'll end up in the `hidden` case and return `userInputNeeded`, as would happen if you attempted a silent update but returned the same status. Either way, the `CachedFileUpdateUI` object will fire its `uiRequested` event, telling the provider app that the system is making the UI visible. The app, in fact, can defer initializing its UI until this event occurs because there's no need to do so for a silent update.

After this, the `fileUpdateRequested` event will fire again with `uiStatus` now set to `visible`. Notice how the code above will have called `request.getDeferral` in this case but has not called its `complete`. We save that step for when the UI has done what it needs to do (and, in fact, we save both the request and the deferral for use from the UI code).

The update UI is responsible for gathering whatever user input is necessary to accomplish the task: collecting credentials, choosing which copy of a file to keep (the local or remote version), allowing for renaming a conflicting file (when updating a remote file), and so forth. When updating a local file, it writes to the `StorageFile` within `request.file` or calls `request.updateLocalFile`; in the remote case it copies data from the local copy in `request.file`.

To complete the update, the UI code then sets `request.status` to `complete` (or any other appropriate code if there's a failure) and calls the deferral's `complete` method. This will change the status of the system-provided buttons along the bottom of the screen, as you can see in Figure 12-19 and Figure 12-21—enabling the OK button and disabling Cancel. In the provider sample, both buttons just execute these two lines for this purpose:

```

fileUpdateRequest.status = Windows.Storage.Provider.FileUpdateStatus.complete;
fileUpdateRequestDeferral.complete();

```

All in all, the interactions between the system and the app for the cached file updater contract are fairly simple and straightforward in themselves: handle the events, copy data around as needed, and update the request status. The real work with this contract is first deciding when to call `setUpdateInformation` and then providing the UI to support updates of local and remote files under the necessary circumstances. This will, of course, involve more interactions with whatever web service or database or whatever else stores the files.

Contacts

The last contract we'll explore in this chapter (whew!) is that of the contact picker. We haven't seen this feature of Windows 8 in action yet. Let's take a look at it first and then explore how the picker is used from one side of the contract and how an provider app fulfills the other side.

A contact, as you probably expect, is information about a person that includes details like name, phone numbers, email addresses, and so forth. An obvious place where you'd need a contact is when composing an email, as shown in Figure 12-22. Here, tapping the + controls to the right of the To and Cc fields will open the contact picker, which defaults to the Windows 8 People app, as shown in Figure 12-23 (its splash screen) and Figure 12-24 (its multiselect picker view, where I have blurred my friends' identities so that they don't start blaming me for unwanted attention!). As we saw with the File Picker UI, the provider app supplies the UI for the middle portion of the screen while the top and bottom bars, the header, and the down-arrow menu control are supplied by Windows using information from the provider app's manifest. (Refer back to Figure 12-17.) Figure 12-25 shows the appearance of the [Contact picker app sample](#) in its provider mode, as well as the menu that allows you to select a different provider (those who have declared themselves as a contact provider).

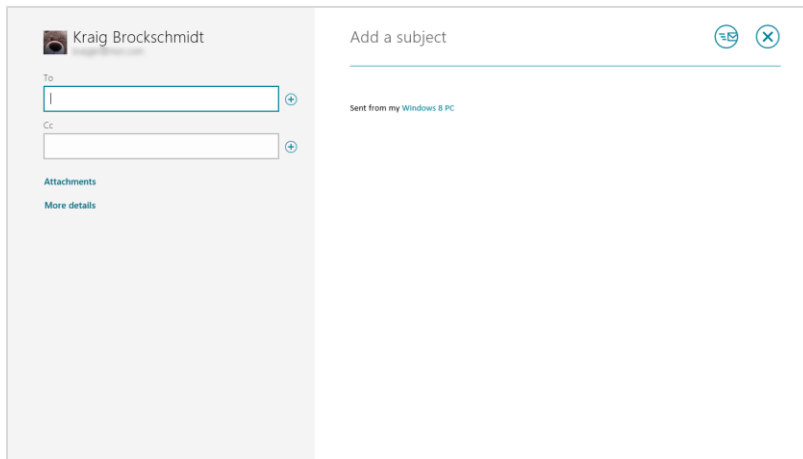


FIGURE 12-22 The Windows 8 mail app uses the contact picker to choose a recipient.

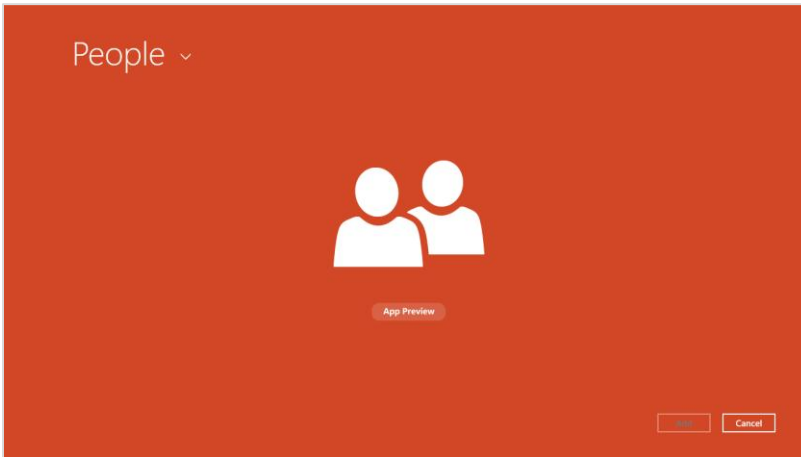


FIGURE 12-23 The Windows 8 People app on startup when launched as a contact provider.

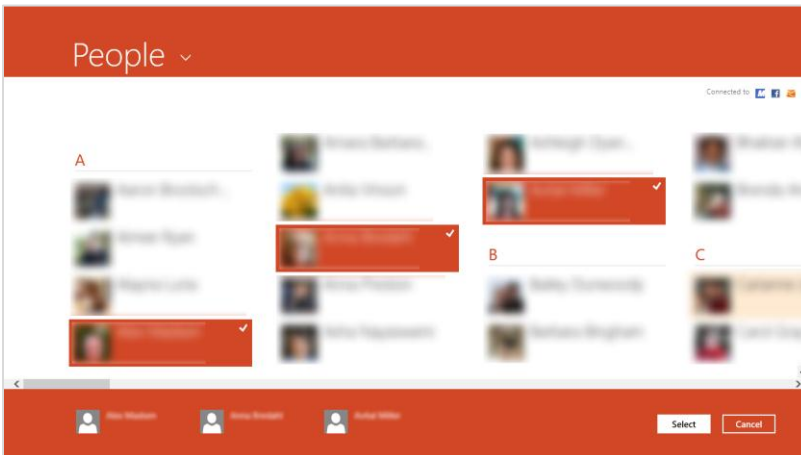


FIGURE 12-24 The picker UI within the Windows 8 People app, shown for multiple selection (with my friends blurred because they're generally not looking for fame amongst developers). The selections are gathered along the bottom in the basket.

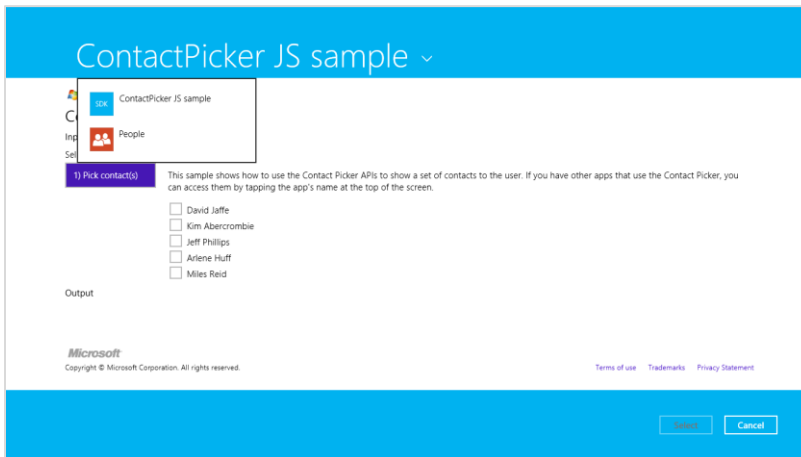


FIGURE 12-25 The Contact Picker sample’s UI when used as a provider, along with the header flyout menu allowing selection of a picker provider.

When I select one or more contacts in any provider app and press the Select button along the bottom of the screen, those contacts are then brought directly back to the first app—Mail in this case. Just as the file picker contract allowed the user to navigate into data surfaced as files by any other app, the contact contract (say that ten times fast!) lets the user easily navigate to people you might select from any other source.

Using the Contact Picker

Contacts as a whole work with the API in the [Windows.ApplicationModel.Contacts](#) namespace. An app that consumes contacts sees each one represented by an instance of the [ContactInformation](#) class, whose properties like [name](#), [phoneNumbers](#), [locations](#), [emails](#), [instantMessages](#), and [customFields](#) give you the contact information, along with the [getThumbnailAsync](#) and [queryCustomFields](#) methods.

Choosing a contract happens through a picker UI much like the file picker, invoked through [Windows.ApplicationModel.Contacts.ContactPicker](#). After creating an instance of this object, you can set the [commitButtonText](#) property to control the first button’s appearance in the picker UI (as with “Select” in the earlier figures). You can also set the [selectionMode](#) property to a value from the [ContactSelectionMode](#) enumeration: either [contact](#) (the default) or [fields](#). In the former case, the whole contact information is returned; in the latter, the picker works against the contents of the picker’s [desiredFields](#). Refer to the documentation on that property for details.

When you’re ready to show the UI, call the picker’s [pickSingleContactAsync](#) or [pickMultipleContactsAsync](#) methods. These provide the completed handler for the async operation with a single [ContactInformation](#) object or a vector of them, respectively. As with the file picker, note that these APIs will throw an exception if called when the app is in snapped view, so you’ll want to avoid doing this.

Picking a single contact and displaying its information is demonstrated in Scenario 1 of the [Contact picker app sample](#) (js/scenarioSingle.js):

```
var picker = new Windows.ApplicationModel.Contacts.ContactPicker();
picker.commitButtonText = "Select";

// Open the picker for the user to select a contact
picker.pickSingleContactAsync().done(function (contact) {
    if (contact !== null) {
        // Consume the contact information...
    }
});
```

Choosing multiple contacts (Scenario 2, js/scenarioMultiple.js) works the same way, just using [pickMultipleContactsAsync](#). In either case, the calling app then applies the [ContactInformation](#) data however it sees fit, such as populating a To or Cc field like the Mail app. However, other than the [name](#) property in that object, which is just a string, its properties have a little more structure, as shown in the following table.

Property	Type	Field Properties and Types
emails phoneNumbers customFields	Vector of ContactField	category (ContactFieldCategory), name (string), type (a ContactFieldType), value (string)
instantMessages	Vector of ContactInstantMessageField	Same as ContactField above plus displayText , launchUri , service , and userName (all strings)
locations	Vector of ContactLocationField	Same as ContactField above plus city , country , postalCode , region , street , and unstructuredAddress (all strings)

Accordingly, the sample consumes a [ContactInformation](#) object as follows, first extracting the individual vector properties:

```
appendFields("Emails:", contact.emails, contactElement);
appendFields("Phone Numbers:", contact.phoneNumbers, contactElement);
appendFields("Addresses:", contact.locations, contactElement);
```

and then enumerating the contents of those vectors and in this case creating elements with their contents. Other apps will, of course, transfer the values to appropriate fields or other parts of the app UI—what's shown here demonstrates processing of the different categories:

```
function appendFields(title, fields, container) {
    // Creates UI for a list of contact fields of the same type, e.g. emails or phones
    fields.forEach(function (field) {
        if (field.value) {
            // Append the title once we have a non-empty contact field
            if (title) {
                container.appendChild(createTextElement("h4", title));
                title = "";
            }

            // Display the category next to the field value
            switch (field.category) {
```

```

        case Windows.ApplicationModel.Contacts.ContactFieldCategory.home:
            container.appendChild(createTextElement("div", field.value + " (home)"));
            break;
        case Windows.ApplicationModel.Contacts.ContactFieldCategory.work:
            container.appendChild(createTextElement("div", field.value + " (work)"));
            break;
        case Windows.ApplicationModel.Contacts.ContactFieldCategory.mobile:
            container.appendChild(createTextElement("div", field.value + " (mobile)"));
            break;
        case Windows.ApplicationModel.Contacts.ContactFieldCategory.other:
            container.appendChild(createTextElement("div", field.value + " (other)"));
            break;
        case Windows.ApplicationModel.Contacts.ContactFieldCategory.none:
        default:
            container.appendChild(createTextElement("div", field.value));
            break;
    }
}
});
}

```

Contact Picker Providers

On the provider side, which is also demonstrated in the Contact picker sample, we see the same pattern as for file picker providers. First, a provider app needs to declare the Contact Picker contract in its manifest, where it indicates the Start page to load within the context of the picker. In the sample, the Start page is `contactPicker.html` that in turn loads `html/contactPickerScenario.html` (with their associated JavaScript files):

The screenshot shows the 'Available Declarations' section with a dropdown menu set to 'Select one...' and an 'Add' button. Below it, the 'Supported Declarations' section lists 'Contact Picker' with a 'Remove' button. To the right, the 'Description' section states: 'Registers the app as a people picker, making contact details in the app available to other Windows 8 apps. Only one instance of this declaration is allowed per app. [More information](#)'. The 'Properties' section includes 'App settings', 'Executable:', 'Entry point:', and 'Start page: contactPicker.html'.

As with the file picker, having a separate Start page means having a separate activated handler, and in this case it looks for the activation kind of `contactPicker` (`js/contactPicker.js`):

```

function activated(eventObject) {
    if (eventObject.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.contactPicker) {
        contactPickerUI = eventObject.detail.contactPickerUI;
        eventObject.setPromise(WinJS.UI.processAll().then(function () {
            // ...
        }));
    }
}

```

The `eventObject.detail` here is a [ContactPickerActivatedEventArgs](#) (these names are long, but

at least they're predictable!). As with all activations, it contains `kind`, `previousExecutionState`, and `splashScreen` properties for the usual purposes. Its `contactPickerUI` property, a [ContactPickerUI](#), then contains the information specific for the contact picker contract:

- The `selectionMode` and `desiredFields` properties as supplied by the calling app.
- Three methods—`addContact`, `removeContact`, and `containsContact`—for managing what's returned to the calling app. These methods correspond to the actions of a typical selection UI, such as that provided by a `ListView`.
- One event, `contactsRemoved`, which informs the provider when the user removes an item from the basket along the bottom of the screen. (Refer back to Figure 12-24.)

Within a provider, each contact is represented by a `Windows.ApplicationModel.Contacts.Contact` object. A provider will create an object for each contact it supplies. In the sample (`js/contactPickerScenario.js`), there's an array called `sampleContacts` that simulates what would more typically come from a database. That array just contains JSON records like this:

```
{
  name: "David Jaffe",
  homeEmail: "david@contoso.com",
  workEmail: "david@cpandl.com",
  workPhone: "",
  homePhone: "248-555-0150",
  mobilePhone: "",
  address: {
    full: "3456 Broadway Ln, Los Angeles, CA",
    street: "",
    city: "",
    state: "",
    zipCode: ""
  },
  id: "761cb6fb-0270-451e-8725-bb575eeb24d5"
},
```

Each record is shown as a check box in the sample's UI (generated in the `createContactUI` function), which is a quick and easy way to show a selectable list of items! Of course, your own provider app will likely use a `ListView` for this purpose; the sample is just trying to keep things simple so that you can see what's happening with the contract itself.

When a contact is selected, the sample's `addContactToBasket` function is called. This is the point at which we create the actual `Contact` object and call `ContactPickerUI.addContact`. The process here for each field follows a chain of other function calls, so let's see how it works for the single *homeEmail* field in the source record, starting with `addContactToBasket` (again in `js/contactPickerScenario.js`). The rest of the field values are handled pretty much the same way:

```
function addContactToBasket(sampleContact) {
  var contact = new Windows.ApplicationModel.Contacts.Contact();
  contact.name = sampleContact.name;
```



```

appendEmail(contact.fields, sampleContact.homeEmail,
    Windows.ApplicationModel.Contacts.ContactFieldCategory.home);

// Add other fields...

// Add the contact to the basket
switch (contactPickerUI.addContact(sampleContact.id, contact)) {
    // Show various messages based on the result, which is of type
    // Windows.ApplicationModel.Contacts.Provider.AddContactResult
}
}

```

As you can see, the *homeEmail* field is passed to a function called `appendEmail`, where the first argument is the vector (`Contact.fields`) in which to add the field and the third parameter is the category (`home`). These are then passed through to another generic function, `appendField`, where the `type` of the field has been thrown into the mix, all of which is used to create a `ContactField` object and add it to the contact:

```

function appendEmail(fields, email, category) {
    // Adds a new email to the contact fields vector
    appendField(fields, email, Windows.ApplicationModel.Contacts.ContactFieldType.email, category);
}

function appendField(fields, value, type, category) {
    // Adds a new field of the desired type, either email or phone number
    if (value) {
        fields.append(new Windows.ApplicationModel.Contacts.ContactField(value, type, category));
    }
}

```

In short, this is essentially how all the fields in a contact are assembled, one bit at a time.

Now, when an item is unselected in the list, it needs to be removed from the basket:

```

function removeContactFromBasket(sampleContact) {
    // Programmatically remove the contact from the basket
    if (contactPickerUI.containsContact(sampleContact.id)) {
        contactPickerUI.removeContact(sampleContact.id);
    }
}

```

Similarly, when the user removes an item from the basket, the contact provider needs to update its selection UI by handling the `contactremoved` event:

```

contactPickerUI.addEventListener("contactremoved", onContactRemoved, false);

function onContactRemoved(e) {
    // Add any code to be called when a contact is removed from the basket by the user
    var contactElement = document.getElementById(e.id);
    var sampleContact = sampleContacts[contactElement.value];
    contactElement.checked = false;
}

```

You'll notice that we haven't said anything about closing the UI, and in fact the `ContactPickerUI` object does not have an event for this. Simply said, when the user selects the commit button (with whatever text the caller provided), it gets back whatever the provider has added to the basket. If the user taps the cancel button, the operation returns a null contact. In both cases, the provider app will be suspended and, if it wasn't running prior to being activated for the contact, terminated.

Do note that as with file picker providers, a contact provider needs to be ready to save its session state when suspended such that it can restore that state when relaunched with `previousExecutionState` set to `terminated`. Although not demonstrated in the sample, a real provider app should save its current selections and viewing position within its list, along with whatever else, to session state and restore that in its `activated` handler when necessary.

What We've Just Learned

- Contracts in Windows 8 provide the ability for any number of apps to extend system functionality as well as extend the functionality of other apps. Through contracts, installing more apps that support them creates a richer overall environment for users.
- The Share contract provides a shortcut means through which data from one app can be sent to another, eliminating many intermediate steps and keeping the user in the context of the same app. A source app packages data it can share when the Share charm is invoked; target apps consume that data, often copying it elsewhere as in an email message, text message, social networking service, and so forth.
- The Share target provides for delayed rendering of items (such as graphics), for long-running operations (such as when it's necessary to upload large data files to a service), and for providing quicklinks to specific targets within the same app (such as frequent email recipients).
- The Search contract provides integration between an app and the Search charm. From the charm users can search the current app as well as any others that support the contract, easily viewing results from other apps without having to manually launch them or switch to them. The search contract allows apps to also provide query suggestions and result suggestions.
- File type and URI scheme associations are how Windows 8 apps can launch other apps. The associations are declared in a supporting app's manifest such that the app can be launched to service the association. URI scheme associations are an excellent means for an app to provide workflow services to others.
- Apps that implement the provider side of the file picker contract appear as choices within the file picker UI. This is how apps can present data sources they manage as if they were part of the local file system, even though they might exist in databases,

online services, or other such locations. To the user, the necessary transport considerations are transparent, and through the cached file updater contract a provider app can also handle synchronization of local and remote copies of the file.

- The contract for Contacts works similarly to the file picker but with information about people. A consuming app can easily invoke the contact picker UI and any number of provider apps can implement the other side of the contract to surface an address book, database, or other source through that UI.



About the Author

Kraig Brockschmidt has worked with Microsoft since 1988, focusing primarily on helping developers through writing, education, public speaking, and direct engagement. Kraig is currently a Senior Program Manager in the Windows Ecosystem team working directly with key partners on building apps for Windows 8 and bringing knowledge gained in that experience to the wider developer community. His other books include *Inside OLE* (two editions), *Mystic Microsoft*, *The Harmonium Handbook*, and *Finding Focus*. His website is www.kraigbrockschmidt.com.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft
Press